

10-30-00

A

MICR-FILM

10/26/00
JCS49 U.S. PTO

09/697994
JCS49 U.S. PTO
10/26/00

UTILITY PATENT APPLICATION TRANSMITTAL

(New Nonprovisional Applications Under 37 CFR § 1.53(b))

Attorney Docket No.
RATLP005C1

TO THE ASSISTANT COMMISSIONER FOR PATENTS:

Transmitted herewith is the patent application of () application identifier or (X) first named inventor, Jeffrey A. Straathof entitled LOAD TEST SYSTEM AND METHOD, for a(n):

- () Original Patent Application.
- (X) Continuing Application (prior application not abandoned):
 - (X) Continuation () Divisional () Continuation-in-part (CIP) of prior Application No. 08/577,278, filed December 22, 1995.
- (X) Please add after the title of the application "This is a
 - (X) Continuation () Divisional () Continuation-in-part (CIP) of Application No. 08/577,278, filed December 22, 1995, which is hereby incorporated by reference."
- () This application claims the benefit of U.S. Provisional Application No. _____, filed _____.

Enclosed are:

- (X) Specification; 32 Total Pages. (X) Drawing(s); 13 Total Sheets.
- () Oath or Declaration:
 - () A Newly Executed Combined Declaration and Power of Attorney:
 - () Signed. () Unsigned. () Partially Signed.
 - (X) A Copy from a Prior Application for Continuation/Divisional (37 CFR § 1.63(d)).
 - () Signed Statement Deleting Inventor(s) Named in the Prior Application. (37 CFR § 163(d)(2)).
- () Power of Attorney. (X) Return Receipt Postcard.
- () Associate Power of Attorney. () A Check in the amount of \$ _____ for the Filing Fee.
- () Preliminary Amendment. (X) Information Disclosure Statement and Form PTO-1449.
- () A Duplicate Copy of this Form for Processing Fee Against Deposit Account.
- () A Certified Copy of Priority Documents (if foreign priority is claimed).
- () Statement(s) of Status as a Small Entity.
- () Statement(s) of Status as a Small Entity Filed in Prior Application, Status Still Proper and Desired.
- (X) Other: Appendix 1 – Microfiche, Appendix 2 – Script development guide, Appendix 3 – Multi-user testing guide, Appendix 4 – Reference guide and other materials

PLEASE DO NOT CHARGE THE FILING FEE AT THIS TIME.

Respectfully submitted,
By: Michael J. Ritter
Michael J. Ritter, Reg. No. 36,653

Date: October 26, 2000

Correspondence Address:

Customer No. 21912
Ritter, Van Pelt and Yi LLP
4906 El Camino Real Suite 205
Los Altos, CA 94022
Phone: 650 903 3500
Fax: 650 903 3501

I hereby certify that this is being deposited with the U.S. Postal Service "Express Mail Post Office to Addressee" service under 37 CFR § 1.10 on the date indicated below and is addressed to:

Assistant Commissioner for Patents
Box Patent Application
Washington, D.C. 20231

By: Jack Limper

Typed Name: Jack Limper

Express Mail Label No.: EL623884751US

Date of Deposit: October 26, 2000

LOAD TEST SYSTEM AND METHOD

COPYRIGHT NOTICE

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the xerographic reproduction by anyone of the patent document or the patent disclosure in exactly the form it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

MICROFICHE APPENDIX

A microfiche appendix of source code including one thousand three hundred nineteen (1319) frames on twenty one (21) sheets is included herewith.

BACKGROUND OF THE INVENTION

The present invention relates to the field of software systems. More specifically, in one embodiment the invention provides an improved method of load testing software and, especially, a system and method for scripting user actions for a load test.

There are a number of different approaches to load testing. For example, live users may be utilized. The system developer hires live users and buys the hardware required (PCs or terminals) to drive the system. This approach, however, is very expensive. In order to cut costs, someone considering this approach usually decides to test using some fraction of the actual number of users that the actual system will be required to support, and to interpolate actual response time figures based on the response time of the fraction. This method fails largely due to inadequate testing of threshold conditions. Also, even if the entire number of proposed users can be seated and instructed to perform the test actions, it

will be impossible to control their rates of input or accurately measure the response times.

Simulations have also been used. The developer computes the theoretical load imposed upon the system and interpolates the response times and theoretical number of users that the system may support. This method has very little basis in reality and provides no confidence that its assumptions are correct or its results accurate.

Canned benchmarks may also be utilized. A number of publicly available benchmark tests are available that exercise a piece of hardware and the operating system that runs it, but this does not allow for the testing of a particular application.

From the above it is seen that an improved system and method for load testing software applications are needed.

SUMMARY OF THE INVENTION

An improved system and method for load testing software applications is provided by virtue of the present invention. The system interjects a Capture Agent that may capture or intercept user interface and application calls in order to generate a higher-level script. The script, along with other scripts, may be compiled and executed to simulate multiple users to load test software applications.

In one embodiment, a method of producing scripts for load testing a software application in a client/server environment, includes the steps of: capturing application calls on the client computer, the application calls including application calls generated in response to the captured user interface calls; recording timing information of the captured application calls; and generating a script from the captured application calls that generates application calls according to the timing information of the captured application calls. Additionally, user interface calls and associated timing information may be captured and incorporated into the script.

In another embodiment, a method of producing scripts for load testing a software application in a client/server environment, includes the steps of: capturing application

calls on the client computer, the captured application calls including references to data stored locally on the client computer; identifying in the captured application calls references to data stored locally on the client computer; 5 accessing the data through the references in the captured application calls; and generating a script from the captured application calls that generates application calls that include the accessed data in place of the references in the captured application calls, the script including the accessed data.

10 In another embodiment, a method of producing scripts for load testing a software application in a client/server environment, includes the steps of: capturing application calls on the client computer, the captured application calls specifying information to be sent to the server computer; 15 translating the captured application calls into source language statements; and generating a script including the source language statements that generates application calls. Additionally, the script may be user edited or compiled to produce an executable load test program.

20 In another embodiment, a method of for load testing a software application in a networked computer system having a first computer (script driver) and a second computer (system under test), includes the steps of: the first computer executing scripts that emulate a plurality of users, the 25 scripts including delays representative of actual user sessions; the first computer sending requests to the second computer over a network; the second computer responding to the requests over the network; and measuring response times of the second computer. Additionally, a third computer may be on the 30 network to display a script directed user session in progress.

A further understanding of the nature and advantages of the inventions herein may be realized by reference to the remaining portions of the specification and the attached drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates an example of a computer system that may be used to execute software of the present invention;

Fig. 2 shows a system block diagram of a typical computer system;

Fig. 3 illustrates a client/server architecture;

Fig. 4 shows a high level flowchart of a typical database application;

Fig. 5 shows the data flow of one aspect of the invention;

Fig. 6 shows a high level flowchart of load testing;

Fig. 7 shows load testing of a system under test;

Fig. 8 shows a flowchart of the process of creating an executable load testing program;

Fig. 9 shows a high level flowchart of the Capture Agent;

Fig. 10 shows a process the Capture Agent performs to generate a script from the capture calls;

Fig. 11 is a typical captured database session;

Fig. 12 shows processing of a database function;

Fig. 13 shows a flowchart of performing load testing;

Fig. 14 shows load testing of a system under test suitable for display or non-display mode; and

Fig. 15 shows a flowchart of a process of emulating a user session from a script.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Definitions:

System - a combination of software and the hardware it runs on.

Load Testing - the process of analyzing the effect of many users on a system.

Response Time - the time between a user-initiated request and the response from the system to that request.

Client/Server - a computer architecture where a networked server computer services client computers that are intelligent, programmable devices.

Load Testing is an essential step in the application development process. It allows a developer or systems

integrator to determine the performance and scalability of a system prior to the deployment of the system by measuring the user-perceived response time. The present invention utilizes user emulation to load test software applications. This process emulates live users by performing the activities of the actual users, preferably in a manner such that the actual system cannot differentiate between the emulated users and the actual users.

Fig. 1 illustrates an example of a computer system that may be used to execute software of the present invention. Fig. 1 shows a computer system 1 which includes a monitor 3, screen 5, cabinet 7, keyboard 9, and mouse 11. Mouse 11 may have one or more buttons such as mouse buttons 13. Cabinet 7 houses a CD-ROM drive 15 or a hard drive (not shown) which may be utilized to store and retrieve software programs incorporating the present invention, data for use with the present invention, and the like. Although a CD-ROM 17 is shown as the removable media, other removable tangible media including floppy disks, tape, and flash memory may be utilized. Cabinet 7 also houses familiar computer components (not shown) such as a processor, memory, and the like.

Fig. 2 shows a system block diagram of a typical computer system. As in Fig. 1, computer system 1 includes monitor 3 and keyboard 9. Computer system 1 further includes subsystems such as a central processor 102, system memory 104, I/O controller 106, display adapter 108, removable disk 112, fixed disk 116, network interface 118, and speaker 120. Other computer systems suitable for use with the present invention may include additional or fewer subsystems. For example, another computer system could include more than one processor 102 (i.e., a multi-processor system) or a cache memory.

Arrows such as 122 represent the system bus architecture of computer system 1. However, these arrows are illustrative of any interconnection scheme serving to link the subsystems. For example, a local bus could be utilized to connect the central processor to the system memory and display adapter. Computer system 1 shown in Fig. 2 is but an example of a computer system suitable for use with the present

invention. Other configurations of subsystems suitable for use with the present invention will be readily apparent to one of ordinary skill in the art.

Fig. 3 illustrates a client/server architecture. A client/server system 130 includes a first computer or server 131 and one or more second computers or clients 150. Typically, the clients 150 are connected to server 131 through a computer network 141, which may be a conventional Local Area Network (LAN). Network 141 includes cabling 145 for connecting the server and each client to the network. The clients themselves may be similar to or the same as computer system 1. Each client typically includes a network connector or adapter 143 for receiving the network cable 145, as is known in the art. Server 131 may also be similar to or the same as computer system 1. Because the server manages multiple resources for the clients, it should preferably include a relatively faster processor, larger mass storage, and more system memory than is found on each client.

Overall operation of the system 130 is directed by a networking operating system 137, which may be stored in the server's system memory. In response to requests from the clients 150, the server 131 provides various network resources and services. For instance, multiple users (e.g., clients A, B and C) may view a database table stored in file server storage 139, while another user (e.g., client E) adds a database record.

The following description will focus on a preferred embodiment of the present invention, where the client computers are IBM compatible computers running Windows 3.1 and the script driver is a UNIX workstation (e.g., from Sun Microsystems, Inc.). The server provides database functions for a database application like those available from Oracle Corporation or Sybase, Inc. The network operating system that provides network communication may be from a vendor such as Novell.

The present invention, however, is not limited to any particular application or any particular environment. Instead, those skilled in the art will find that the teachings of the present invention may be advantageously applied to a variety of

other applications including spreadsheet applications, word processors, and the like. Moreover, the present invention may be embodied on a variety of different platforms, including Macintosh, NextStep, and the like. Therefore, the description that follows is for purposes of illustration and not limitation.

One difficulty encountered in user emulation is the problem is how to develop a test script that accurately represents a user's activities and rates of input to the system being tested. This problem is complicated when the system to be tested is a part of a client/server application, where the client part of the system is located on a personal computer (PC) and the server part on another, usually larger, host.

Systems provided by this invention implement a method of capturing user activities that are part of a client/server system. This method works in a way that accurately records the user's rates of input, client delays caused by local processing of data, and all other inputs necessary to correctly replay and replicate that user's activities to perform a load test. Although the particular implementation is specifically developed to capture particular commercially available database user's activities (Oracle, Sybase, etc), it can be modified for another database or for another type of client/server application. It can also be modified for another client side operating system such as OS/2, Windows '95, or Windows NT.

With the invention, a single machine may be used to accurately simulate hundreds of users. The system creates scripts that represent actual users and their daily, often disparate, operations. With the Capture Agent, the system records user activities, including keystrokes, mouse movements and SQL requests, to create emulation scripts. The system then arranges a mix of scripts that represent actual users. The system reveals if a software application or system under test works. Before deployment, one can correct common difficulties that emerge during the application development process. The system may optionally chart the time one waits for screen responses, and may find hidden bugs and bottlenecks.

In a preferred embodiment, the Capture Agent captures Windows and SQL Application Programming Interface (API) calls. These user interface and application calls may be captured or intercepted by an in-memory replacement of API call addresses with Capture Agent function addresses, by a renaming of the API library on disk so that the Capture Agent is called in place of the API library which then calls the renamed library, hooks, or other mechanisms. As will be discussed in reference to Fig. 5, the Capture Agent monitors the API calls and generates a script that encapsulates the calls into a form that may be executed on another host (or script driver). This is unique in that it is not merely a trace of the API calls. The calls to the API are monitored and re-written into a higher-level or source language that allows the data and a representation of the API calls to be recorded into a procedural script that can be compiled and executed. Among other things, this requires references to program variable addresses to be resolved by the Capture Agent and the data referred to to be recorded in the script.

Another unique part of this process is that while the application calls (e.g., SQL API calls) are being monitored, the user interface calls (e.g., Windows API calls) of the operating system are also being monitored so that the user's activities may be recorded. These activities or events include mouse motions, button presses, key presses, as well as user-generated delays. The process differentiates between delays caused by the user (due to a user pause), delays caused by the client-side of the application (due to some internal processing of data, for example), and delays caused by the server-side of the application. Since the goal is to accurately emulate all of the activities of a client, these delays are important to the emulation of the user. Discarding delays caused by the internal processing of data (for example) will cause a higher load to be imposed on the server due to faster arrival times of requests.

Fig. 4 shows a high level flowchart of a typical database application. The database application may be from such vendors as Oracle Corporation or Sybase, Inc. Although the client/server application described is a database

application, other applications may be utilized with the present invention. For example, spreadsheet applications, word processors, or any other client/server application.

Once a database application is running, the system receives user input at step 202. The user input may be any type of user input like typing characters on the keyboard or moving the mouse and "clicking" (depressing a mouse button) on a menu selection in a Graphical User Interface (GUI). The operating system typically receives these user events and makes a user interface call to the appropriate application which, in this case, will assumed to be the database application.

At step 204, the database application receives the user interface call from the operating system. The database application then processes the call to determine what action, if any, should be taken. The following describes just a few of the possible user interface calls that may be received. The calls are not intended to be an exhaustive list and may vary from application to application. Nevertheless, the calls are intended to aid the reader's understanding of a typical database application.

At step 206, the database application determined that the user interface call is a request to logon to the database application. A logon typically includes the user's name and a password. The database application then verifies and stores the logon information.

At step 208, the database application received a user interface call requesting to insert data into the database. After the database application verifies the request, the database application makes an application call to put the data in the database at step 210. The application call is sent over the network to the database server.

At step 212, the database application received a user interface call requesting to fetch data from the database. The database application makes an appropriate application call to get the requested data from the database at step 214. The application call is sent over the network to the database server.

At step 216, the database application determined that the user interface call is a request to logoff the database application. After each of the following user interface calls the system receives more user input at step 202 which may be passed on to the database application. However, if the user interface call specifies that the user desires to exit the database application, the database application exits at step 218.

Fig. 5 shows the data flow of one aspect of the invention. A client computer 252 is linked to a server computer 254 over a network. In the embodiment shown, the client is running Windows as an operating system and the database application is an SQL database like Oracle. A user interacts with the client through a user interface 256, here shown as Windows. As the user types characters on the keyboard or operates the GUI, these user events generate the appropriate user interface call in the form of a Windows API call to a client application 258. The client application is a front-end for the database application, which is shown as an SQL server. The front-end client application utilizes the processing power of the client computer in order to offload some of the processing required from the database server. Thus, the database application has a client-side application and a server-side application. Typically, the client application is optimized for user interaction and the server application of the database application is optimized for servicing multiple users to take advantage of the client/server architecture.

The client application receives user interface calls and sends the appropriate application calls in the form of an SQL API call to a network transport layer 260. The network transport layer is typically a driver that formats the call for transmission over the network to the server. Similarly, the network transport layer receives data from the server sent to the client.

An important aspect of the present invention is the Capture Agent. A Capture Agent 262 captures one or both of the Windows and SQL API calls during a user session. The Capture Agent not only intercepts the user interface and application

calls, the Capture Agent records timing information regarding when the calls were sent. This allows the Capture Agent to generate a script 264 to emulate the user session including the speed in which the user input information and the speed in which the client computer responded locally.

The Capture Agent may intercept user interface and application calls any number of ways known in the art. For example, the Windows API calls are intercepted by a Windows API call "hook" which is provided for this purpose. The SQL API calls may be intercepted by renaming the "real" calls so that the SQL API calls go to the Capture Agent. After the Capture Agent intercepts the SQL calls, the "real" SQL call is then sent so that the user session may continue.

The script emulates a user session by being able to reproduce the user and application delays. This is more comprehensive approach than merely monitoring the network traffic. Preferably, the script is written in a source language meaning a programming language which includes human-readable program statements. This allows the scripts to be more easily edited to vary the captured user session and add control constructs. In a preferred embodiment, the script includes program statements in the C programming language.

Fig. 6 shows a high level flowchart of load testing. Many of the steps shown are optional and many of the steps are more thoroughly discussed in the specification that follows. However, this high level flowchart is provided to give an overall view before discussing more detailed aspects of the present invention.

A user session is begun on a client so that the user interface and application calls may be captured at step 302. The calls are captured by the captured agent which records timing information regarding the captured calls at step 304. During or after the user session, a script is generated at step 306 that is capable of directing emulation of the user session. The Capture Agent generates the script which preferably includes source language statements and the timing information of the capture user interface and application calls.

At step 308, the script may be edited. Although this step is optional, it may be useful to edit the script in order to enhance the script (e.g., add loops) or modify the data. For example, if a script of a typical user session that adds a database record is captured. It may be beneficial to edit the script so that the script adds different database records when multiple copies of the script are executing. Thus, if a user session adds a database record for an employee named John Smith, the script may be edited to add a record for an employee from a data file or a random name. In this manner, when the script is utilized to emulate, for example, a hundred users, all one hundred users will not attempt to add the same database record. Not only would this run the risk of causing an error, it would not adequately emulate realistic multiple user sessions.

The script or scripts are compiled at step 310. In a preferred embodiment, the scripts include source language statements and may be compiled into executable programs that emulate user sessions. Load testing may be performed with a single script or with multiple scripts. Typically, however, multiple scripts are utilized to simulate tens or hundreds of users.

At step 312, load testing of an application or system under test is performed utilizing the compiled scripts. The compiled scripts are run by a script driver which is a computer connected to the server over a network similar to or the same as the network that will be utilized to connect the clients to the server in actual operation.

Fig. 7 shows load testing of a system under test. A script driver 352 is connected to a server 354 by a network. In one embodiment, the script driver is a relatively fast computer, workstation or mainframe running the UNIX operating system. For example, the script driver may be a workstation from Sun Microsystems, Inc. which when running UNIX provides multitasking capabilities which are utilized to emulate multiple user sessions.

The script driver may store multiple compiled scripts 356. The scripts are run on the script driver as multiple

may include user interface calls, application calls, and timing information of the calls. The Capture Agent is stopped at step 406. The Capture Agent is an important aspect of the present invention and its operation will be more fully described in reference to Figs. 9-12.

The script may be edited at step 408. As mentioned previously, the script may be edited to insert program control statements like loops or to alter data so that the data more accurately represents user sessions. Additionally, the script may be edited for any number of reasons. Because the script contains source language statements in preferred embodiments, the script is not only human-readable but provides the power and capabilities of the source language (e.g., the C programming language). Additionally, captured application calls for different applications from different vendors may be translated into the same source language.

After the script is edited, the script is transferred to the script driver at step 410. The script is typically sent over the network to the script driver but may be input into the script driver utilizing other means like floppy disks.

At step 412, the script is compiled on the script driver. Before the script is compiled, it may be run through a script preprocessor that adds source language statements or other information to the script to make it more suitable for compiling. For example, in a preferred embodiment where the source language is the C programming language, the scripts do not contain compiler preprocessor statements like "#include" or keywords like the curly braces "{" and "}". The script preprocessor adds to or otherwise modifies the script so that it is ready to be compiled.

The script or scripts are utilized to form an executable load testing program at step 414. If a single script is to be utilized to emulate a single user session, the script itself represents an executable load testing program for the user. However, typically the load testing program simulates multiple users running different user sessions. A Mix program allows more than one script to be combined to simulate multiple users. The definition of variables for the

processes so that transactions 358 are sent to the server. Because of the timing information in the scripts, the script driver may faithfully emulate the actions of multiple users. Therefore, the server is operating as if multiple users are
5 utilizing the database application from client computers on the network. The server responds by sending results 360 to the script driver in response to the transactions.

Although Fig. 7 shows a minimal system where only a script driver and the system under test are used for load
10 testing, load testing may be performed with any number of client computers on the network "between" the script driver and the server so that the clients display user sessions as they are run. This aspect of the present invention is called "display mode" and will be discussed in more detail in
15 reference to Figs. 10, 13 and 14.

Fig. 8 shows a flowchart of the process of creating an executable load testing program. The executable load testing program is the program that operates on the script driver to emulate one or typically many users. The process
20 steps are performed on different computers including the clients and the script drivers. Although these steps may be performed sequentially on a network including the script driver, clients and server. Many of the steps may be performed on different networks, at different locations, and at different
25 times. For example, the generation of scripts does not require the script driver and may be performed on one or numerous client computers. As another example, the scripts are described as being edited on the client but they may be edited, if at all, on any computer that provides script editing
30 capabilities (typically ASCII editing). Therefore, the actual process of executable load testing program will vary according to many factors which will be readily apparent to those of skill in the art.

At step 402, the Capture Agent is initialized on a
35 client computer. The Capture Agent is directed by a program to capture user interface and application calls to generate a script. The Capture Agent generates a script on the client computer according to the user session at step 404. The script

load testing program like the number of users, the scripts to be utilized for each user and any delays between the execution of scripts may be specified. In a preferred embodiment, the Mix program may be run in a batch mode or interactively.

As an example, a very simple table may be created in a file called "4user.tab" to include the following:

```
user1, script1
user2, script1
user3, script1
user4, script2
```

This table specifies that there are four users and that three of the users will be emulated by script1 and one user will be emulated by script2 (where "script1" and "script2" are the compiled script names). Although the compiled scripts may be started interactively either individually or utilizing a table file like 4user.tab, it is oftentimes easier to utilize a batch file.

Multiple batch files may be created that utilize the 4user.tab table file. For example, a batch file may be created that starts all the users specified by the scripts as follows:

```
use 4user.tab
start all
wait
quit
```

The first statement directs the Mix program to utilize the 4user.tab table. The next three statements direct the Mix program to start all the scripts in the table and wait for them to finish before quitting.

Another batch file may be created that utilizes the 4user.tab table file which starts three of the users specified by the scripts five seconds apart thirty seconds into the mix session. The batch file would be as follows:

```
use 4user.tab
at 30 start user1
at 35 start user2
at 40 start user4
wait
quit
```

As before, the first statement directs the Mix program to utilize the 4user.tab table. The next three statements specify the number of seconds from the beginning of the mix session when user1, user2 and user4 should be started. Thus, thirty

seconds after the mix session begins, user1, user2 and user4 will be started five seconds apart. The wait and quit statements direct the Mix program to wait for the started users to finish before quitting.

5 The preceding were just a couple simple examples of the power that the Mix program provides. When used interactively, the Mix program allows simple load tests to be quickly initiated. The batch file capability allows more complicated load tests to be created and saved for future use.

10 Fig. 9 shows a high level flowchart of the Capture Agent. Capture is initiated by a user at step 402 of Fig. 8. For ease of reference, this same step is shown as step 452. After the user starts capture, the Capture Agent is launched at step 453. The Capture Agent is directed by a computer program called the Capture program.

15 At step 454, the Capture Agent may check the license on the script driver. This step is optional but may be utilized to ensure that the software on the script driver is licensed software. In one embodiment, the script driver software is licensed only for particular computers. The Capture Agent then verifies that the license on the script driver matches the script driver computer. If the license is not verified, the Capture Agent will not proceed.

20 The Capture Agent installs hooks to capture user interactions and database functions at step 456. Thus, the Capture Agent installs or sets up whatever hooks or interception mechanisms are needed to capture user interface and application calls. The "hook" API call may be utilized to capture Windows API calls and renaming SQL API calls may be
25 utilized to capture SQL API calls. Other mechanisms may be utilized depending on the nature of the computer architecture, operating system, network operating system, and application to be tested.

30 At step 458, the Capture Agent monitors user interface and application calls to generate a script. The Capture Agent captures or intercepts the calls and timing information of when the calls were sent. In this manner, the generated script is able to reproduce the user session
35

Capture is stopped by a user at step 406 of Fig. 8.

Fig. 10 shows a process the Capture Agent performs to generate a script from the capture calls. At step 502, the user interface and application calls are captured. Timing information regarding when the calls were captured is recorded at 504. The timing information may include or be used to calculate think time for user interface calls and application processing time for application calls.

The term "application processing time" is used to refer to the time starting when a user interface call is received from the user and ending when the client application responds locally, for example by redrawing a window on the display. The application processing time does not refer to the time that the server takes to process an SQL call. For example, if it takes 0.4 seconds for the client to bring up a

window used for fetching data, then a application processing time of 0.4 second may be recorded.

The think and application processing times are utilized to emulate a user session in "non-display mode". In non-display mode, the script driver communicates directly with the server as shown in Fig. 7. Because a client computer is not performing the client application functions, the user input to the client application is not utilized. Instead, the think and application processing times are utilized.

In display mode, at least one client computer accepts the user input to the client application and reproduces the screen displays. This arrangement is shown in Fig. 14. Thus, in display mode, the script driver communicates with the server through a client computer which may be very useful during load testing of a system. Display mode or non-display refers to a single emulated user session or script. Consequently, some scripts may be run in display mode while other scripts are run in non-display mode. Of course, display mode requires that a client be available so the actual number may be limited by the hardware available.

In order to illustrate the effect of display mode and non-display mode on the script, the following script segment will be analyzed:

```

...
Think(2.026);
LeftButtonPress(459, 616);
AppWait(0.22);
WindowRcv("CwPtPt");
...

```

The Think statement indicates the user took 2.026 seconds to click on the left mouse button as indicated by the following LeftButtonPress statement. The LeftButtonPress statement indicates that the user pressed the left mouse button at those coordinates. The AppWait statement indicates the client computer took 0.22 seconds to redraw a window as indicated by the following WindowRcv statement. The WindowRcv statement indicates that certain windowing events were taken in response to the LeftButtonPress. The parameters of the WindowRcv function are standard Windows two-letter pneumonics (e.g., "Cw" means create window and "Pt" means paint).

When the above script is run in display mode, the LeftButtonPress command will be sent to a client computer after the indicated think time and the WindowRcv command instructs the script driver to wait until the indicated windowing events occur. As the script is actually driving a client computer and waiting for the client application to perform, the AppWait function is ignored.

When the above script is run in non-display mode, only the Think and AppWait statements are executed. The script driver waits the appropriate times to emulate the user session but the user interface calls and any responses to them are not utilized. All the scripts in Fig. 7 would be run in non-display mode as there is no client to display a script in progress. In Fig. 14, scripts may be run in display and non-display mode as there is at least one client to display a script in progress.

Additionally, the Capture Agent may be instructed to only capture the application calls (e.g., SQL API calls). This may be done to conserve space in the script files. Thus, the above script would not include the LeftButtonPress statement. However, the Think statement would still be present so that the script delays time associated with user input time. The script may then only be run in non-display mode as the user interface calls are not present in the script.

Although a user may physically operate a client computer while generating a script for a user session, the present invention may also be utilized with a GUI tester. A GUI tester is a program that takes over the inputs to an application in order to test it. The Capture Agent may be utilized with a GUI tester so that the GUI tester operates the client application and the Capture Agent generates a script. Thus, the present invention may be used in conjunction with GUI testers and does not require that a user operate the client to generate a script for a user session. Accordingly, the terms "user session", "user interface call" and the like, do not imply that a living user must be operating the client.

Still referring to Fig. 10, pointers in the application calls to client data are dereferenced at step 506.

Many application calls include references or pointers to data that is stored locally on the client. The Capture Agent dereferences these pointers (accesses the data referenced) and places the data in the script. For example, the following is a script segment including SQL calls:

```

...
Open(LOG1, CUR1)
Parse(CUR1, "INSERT INTO CUSTOMERS (ID, FIRST_NAME,
LAST_NAME, ADDRESS_LINE_1, ADDRESS_LINE_2,
ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER,
COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS) VALUES
(:id, :fname, :lname, :addr1, :addr2, :addr3, :phno,
:faxno, :cytd, :bal, :comm)");
CSset(CUR1, MAXARRSIZE, 1);
Bind(CUR1, ":id", STRING, 40);
Bind(CUR1, ":fname", STRING, 21);
Bind(CUR1, ":lname", STRING, 21);
Bind(CUR1, ":addr1", STRING, 21);
Bind(CUR1, ":addr2", STRING, 21);
Bind(CUR1, ":addr3", STRING, 21);
Bind(CUR1, ":phno", STRING, 16);
Bind(CUR1, ":faxno", STRING, 16);
Bind(CUR1, ":cytd", STRING, 40);
Bind(CUR1, ":bal", STRING, 40);
Bind(CUR1, ":comm", STRING, 241);
Data(CUR1, "555|John|Smith|123 Elm St.|Anytown, MD
12345|USA|555-555|555-5555|9000|5000|No comments");
Exec(CUR1);
Close(CUR1);
...

```

The above script include source language statements in the C programming language. The statements add a new data record John Smith into CUSTOMERS. The statements correspond very closely to actual SQL API calls. However, in actual SQL API calls, the Bind statements may have pointers to the data for John Smith on the client computer (e.g., the Bind statements may have pointers as parameters).

The Capture Agent identifies the pointers, dereferences the pointers to access the data, and automatically generates the Data statement which includes the accessed data.

In this way, the script driver is able to replicate the user session without requiring access to the client's memory, which is typically unavailable during load testing. Another advantage is that by dereferencing the pointers and placing the accessed data in the script, the script may be more easily edited so that each copy of the script will add different customers to the database when run. This may be done by filling in the data statement from a file or using some kind of random data.

At step 508, the script is generated. The script may be generated during and after a user session. In other words, the script may be generated at the end of a captured user session, during the captured user session, or both. The script is typically saved onto the hard drive of the client computer.

Fig. 11 is a typical captured database session. Once the client application of the database application is running on the client, a Think timer is started at step 552. The user inputs data at step 553 until a terminating character (e.g., the enter key) or action (e.g., clicking a mouse button) has been taken.

After the user is finished inputting data, the Think timer is stopped at 554. The user input in the associated user interface call is recorded at step 555. If the user input indicates the user wants to quit the application at step 556, the application terminates. Otherwise, an Application Processing timer is started at step 558.

At step 559, the user input in the form of the user interface call is sent to the client application. Once the client application receives the user interface call, the application performs whatever processing is required at step 560. For example, the user interface call may specify to perform a database function as shown in step 562 or to display results in step 564. The database function typically requires the client application to send a request to the server via the network. The processing of a database function will be described in more detail in reference to Fig. 12.

The client application continues to process the user interface call until all the processing has been done. The processing has been completed at step 566.

Fig. 12 shows processing of a database function. At step 602, the client application calls a database function (i.e., an application call). In a preferred embodiment, the database function is in the form of an SQL API call and the database function calls a substitute database function so that the Capture Agent can capture or intercept the call. The substitute database function is called at step 604.

The Application Processing timer is stopped at step 606. Information being sent to the database server in the database function is saved at step 608. The information includes the database call and any data that is referenced through pointers in the call. At step 610, the real database function is called.

At step 612, a determination is made as to whether the database function resulted in an error from the server. For example, the error may be caused by an incorrect or ill-formed database function call. However, when capturing a user session, a user may generate an error. In order to reproduce the user session, the script should also do the same actions to generate the error. These errors though are expected so the script sets an error condition (e.g., a flag) to CONTINUE at step 618. When the script later encounters this error during execution, the script will check the error condition to see if it is set to CONTINUE. If the error condition is set to CONTINUE, the script knows that the error was expected and continues.

If the database function did not result in an error from the server, the error condition is set to EXIT at step 620 which instructs the script to exit if it encounters an error because it is unexpected. The error condition does not have to be set for each database function but may only be set when the error condition needs to be changed. Typically, most database functions will not generate an error so the error condition will usually be set to EXIT. The script will then exit when an unexpected error occurs during script execution.

At step 622, data coming back from the database server (e.g., results) are saved. Script statements are then generated at step 624. Once the script statements are generated, the Application Processing timer is started at step 626 and control is returned to the client application at step 628.

Fig. 13 shows a flowchart of performing load testing. The flowchart assumes that there are already compiled scripts resident on the script server. At step 652, a determination is made whether the load test should be performed in display mode. In order for the load test to be performed in display mode, there should be at least one client computer on the network as shown in Fig. 14. For each client that will display the script in progress, a Display program is started on the client at step 654.

The Display program is designed to accept input from the script driver to simulate a live user. The input is generated by the script and was typically captured from a previous user session.

At step 656, the Mix program is run. The Mix program allows, either interactively or in batch files, more than one script to be executed on the script driver. The Mix program is optional but is typically run to simulate multiple users interacting with the server. A flag or parameter is set for each script that will be executed in display mode so that the script driver sends the appropriate inputs to the client.

Each script that executes on the script driver creates a log file at step 660. A report may be generated from the log files which includes information such as server response time and system throughput. In a preferred embodiment, the log files are ASCII files on the server so that third party statistical and graphics programs may be utilized to create custom reports.

At step 664, a determination may be made whether to rerun another load test. For simplicity, the flowchart shows a return to step 656 to run the Mix program and possibly select a different set of users, number of users, user start times, and the like. However, this should not imply that a return to many

of the other steps is not possible. For example, a different client may be selected to run the display program at step 654 or a new user session may be captured on the client at step 402 of Fig. 8.

Fig. 14 shows load testing of a system under test suitable for display or non-display mode. A script driver 702, clients 704 and a server 706 are connected by a network. The script driver may store multiple compiled scripts 708. Scripts that are run in display mode send user interface calls 710 to one of the clients 704. The clients then display the screen displays that occur during the emulated user session and also send transactions 712 to the server. The server sends results 714 to the clients in response transactions 712. The clients 704 forward results 714 to the script driver.

Scripts that are run in non-display mode send transactions 718 directly to the server. the server sends results 720 to the script driver in response to transactions 720. Transactions 712 and 718 do not differ except for the source of origin of the transactions being the script driver or clients, respectively.

Fig. 15 shows a flowchart of a process of emulating a user session from a script. The scripts execute on the script driver with each script representing a user session that was typically captured by the Capture Agent. In a preferred embodiment, the scripts are compiled and include source language statements so the execution of a script is the execution of a computer program. The scripts may contain different control statements including loops and branches so there is no "one" process flow of a script. The script process in Fig. 15 is presented to illustrate a simple script.

At step 752, the script starts and begins executing the compiled source language statements. If there are more functions or statements at step 754, the functions are executed.

A database function is specified at step 756. The database function is translated into the appropriate application call (e.g., SQL API call) which is sent to the server via a network transport layer.

User processing is specified at step 760. The user processing may include Think delays, Pointer (e.g., mouse) delays or Typerate (speed of typing) delays. The script may have any of these delays globally defined at the beginning of the script. If the script is executed in display mode, the specified user interface call or calls will also be executed or sent at step 762 after the delays.

Application processing is specified at step 764. An application processing delay (e.g., AppWait) will be performed at step 766 if the script is executed in non-display mode. If the script is executed in display mode, the script will wait for the client to return from generating the window displays.

Once all the functions or statements in the script have been executed, the script creates a log file at step 768. The log file may be created and written to as the script executes but the step is shown at the end for simplicity. After the log file has been created, the script exits at step 770.

Appendix 1 includes source code for a computer software product that includes the Capture Agent that captures user interface and application calls as well as other aspects of the present invention. Appendix 2 is a script development guide. Appendix 3 is a multi-user testing guide. Appendix 4 is a reference guide as well as other related materials.

The above description is illustrative and not restrictive. Many variations of the invention will become apparent to those of skill in the art upon review of this disclosure. Merely by way of example the invention is illustrated with regard to database applications, but the invention is not so limited. The scope of the invention should, therefore, be determined not with reference to the above description, but instead should be determined with reference to the appended claims along with their full scope of equivalents.

WHAT IS CLAIMED IS:

1 1. In a computer system having a server computer
2 and a client computer, a method of producing scripts for load
3 testing a software application, comprising the steps of:
4 capturing application calls on the client computer,
5 the application calls including application calls generated in
6 response to user interactions;
7 recording timing information of the captured
8 application calls; and
9 generating a script from the captured application
10 calls that generates application calls according to the timing
11 information of the captured application calls.

1 2. The method of claim 1, setting a timer to
2 determine timing information of the captured application calls.

1 3. The method of claim 1, further comprising the
2 steps of:
3 capturing user interface calls on the client
4 computer;
5 recording timing information of the captured user
6 interface calls; and
7 generating the script from the captured user
8 interface calls that generates user interface calls according
9 to the timing information of the captured user interface calls.

1 4. The method of claim 3, setting a timer to
2 determine timing information of the captured user interface
3 calls.

1 5. The method of claim 1, further comprising the
2 step of setting a flag in the script if an error is generated
3 during script generation.

1 6. The method of claim 5, further comprising the
2 step of ignoring an error generated during script execution if
3 the flag indicates the script was also generated during script
4 generation.

1 7. The method of claim 1, further comprising the
2 step of executing a plurality of scripts to emulate a plurality
3 of users.

1 8. The method of claim 1, further comprising the
2 step of creating a report of response times of the server
3 computer.

1 9. The method of claim 1, wherein the captured user
2 interface calls are Windows API calls.

1 10. The method of claim 1, wherein the captured
2 application calls are SQL API calls.

1 11. A computer program that produces scripts for
2 load testing a software application, comprising:
3 computer readable code that captures application
4 calls on a client computer, the application calls including
5 application calls generated in response to user interactions;
6 computer readable code that records timing
7 information of the captured application calls; and
8 computer readable code that generates a script from
9 the captured application calls that generates application calls
10 according to the timing information of the captured application
11 calls;

12 wherein the computer readable codes are stored on a
13 tangible medium.

12. In a computer system having a server computer and a client computer, a method of producing scripts for load testing a software application, comprising the steps of:

capturing application calls on the client computer, the captured application calls including references to data stored locally on the client computer;

identifying in the captured application calls references to data stored locally on the client computer;

accessing the data through the references in the captured application calls; and

generating a script from the captured application calls that generates application calls that include the accessed data in place of the references in the captured application calls, the script including the accessed data.

13. The method of claim 12, wherein the references in the captured application calls are pointers.

14. The method of claim 12, wherein the captured application calls are SQL API calls.

15. The method of claim 12, wherein the script includes source language statements.

16. The method of claim 15, further comprising the step of user editing the script.

17. The method of claim 15, further comprising the step of compiling the script into an executable load test program.

18. The method of claim 12, further comprising the step load testing a system utilizing a plurality of scripts.

1 19. A computer program that produces scripts for
2 load testing a software application, comprising:

3 computer readable code that captures application
4 calls on the client computer, the captured application calls
5 including references to data stored locally on the client
6 computer;

7 computer readable code that identifies in the
8 captured application calls references to data stored locally on
9 the client computer;

10 computer readable code that accesses the data through
11 the references in the captured application calls; and

12 computer readable code that generates a script from
13 the captured application calls that generates application calls
14 that include the accessed data in place of the references in
15 the captured application calls, the script including the
16 accessed data;

17 wherein the computer readable codes are stored on a
18 tangible medium.

1 20. In a computer system having a server computer
2 and a client computer, a method of producing scripts for load
3 testing a software application, comprising the steps of:

4 capturing application calls on the client computer,
5 the captured application calls specifying information to be
6 sent to the server computer;

7 translating the captured application calls into
8 source language statements; and

9 generating a script including the source language
10 statements that generates application calls.

1 21. The method of claim 20, further comprising the
2 step of translating captured application calls for different
3 applications into a same source language.

1 22. The method of claim 20, further comprising the
2 step of capturing user interface calls on the client computer.

1 23. The method of claim 22, further comprising the
2 step of translating the captured user interface calls into
3 source language statements.

1 24. The method of claim 23, wherein the script
2 generates user interface calls.

1 25. The method of claim 20, wherein the source
2 language statements are in the C programming language.

1 26. A computer program that produces scripts for
2 load testing a software application, comprising:

3 computer readable code that captures application
4 calls on the client computer, the captured application calls
5 specifying information to be sent to the server computer;

6 computer readable code that translates the captured
7 application calls into source language statements; and

8 computer readable code that generates a script
9 including the source language statements that generates
10 application calls;

11 wherein the computer readable codes are stored on a
12 tangible medium.

1 27. In a networked computer system having a first
2 computer (script driver) and a second computer (system under
3 test), a method of for load testing a software application,
4 comprising the steps of:

5 the first computer executing scripts that emulate a
6 plurality of users, the scripts including delays representative
7 of actual user sessions;

8 the first computer sending requests to the second
9 computer over a network;

10 the second computer responding to the requests over
11 the network; and

12 measuring response times of the second computer.

1 29. The method of claim 27, further comprising the
2 step of selecting a display mode where a third computer on the
3 network displays a script directed user session in progress.

ABSTRACT

5

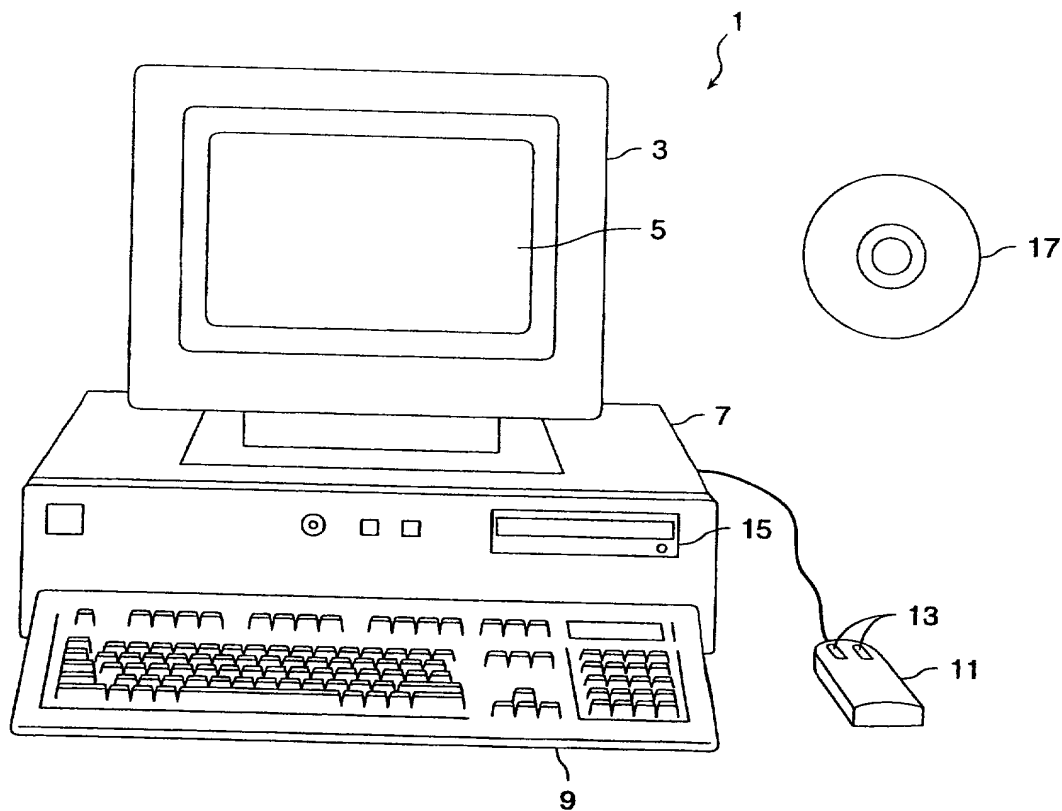


FIG. 1

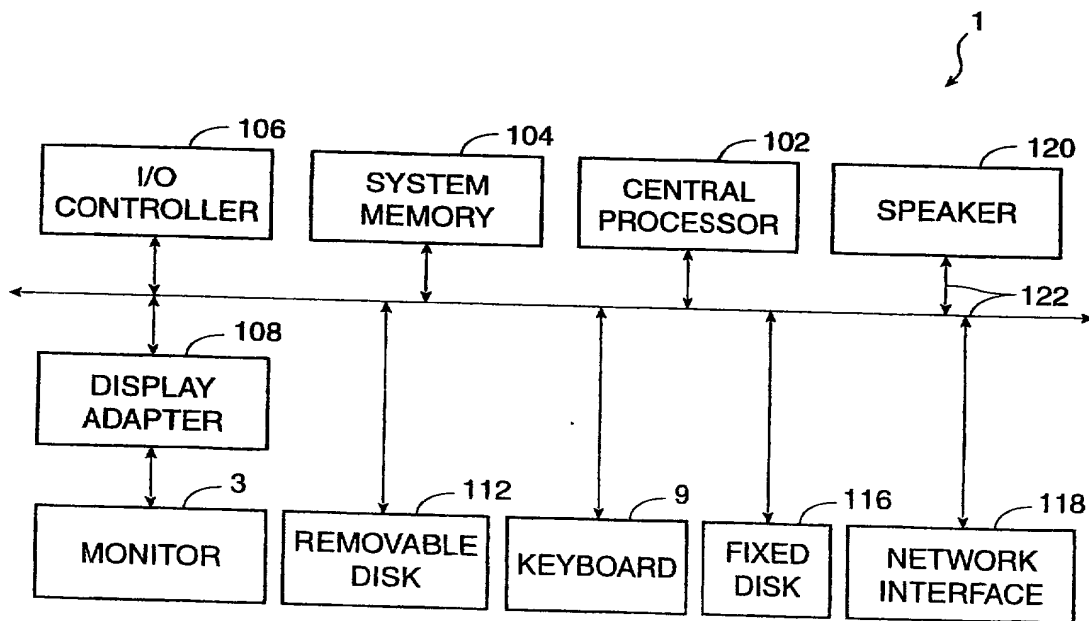


FIG. 2

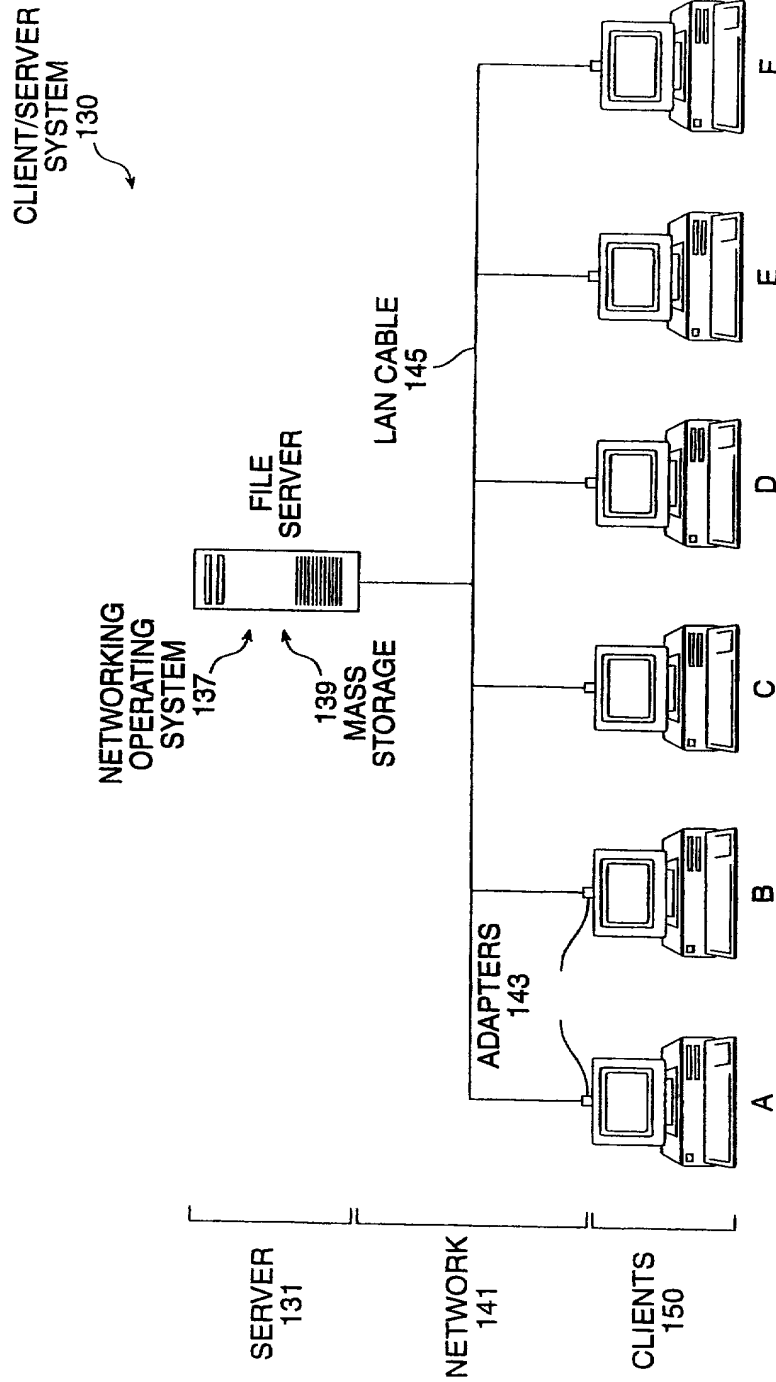


FIG. 3

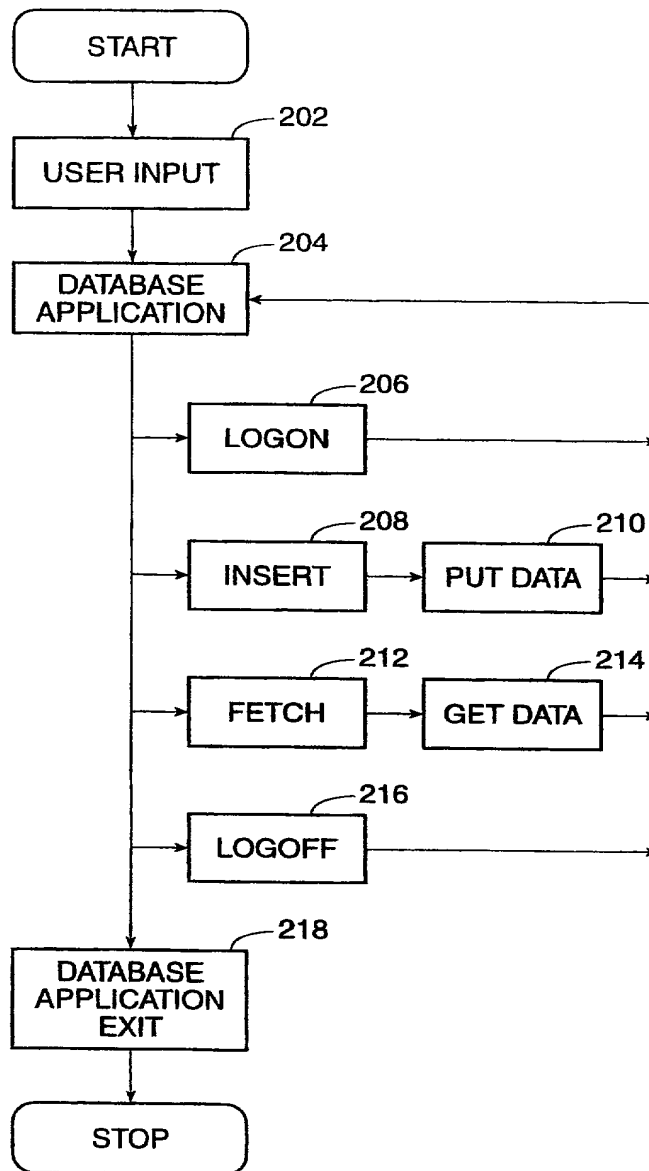


FIG. 4

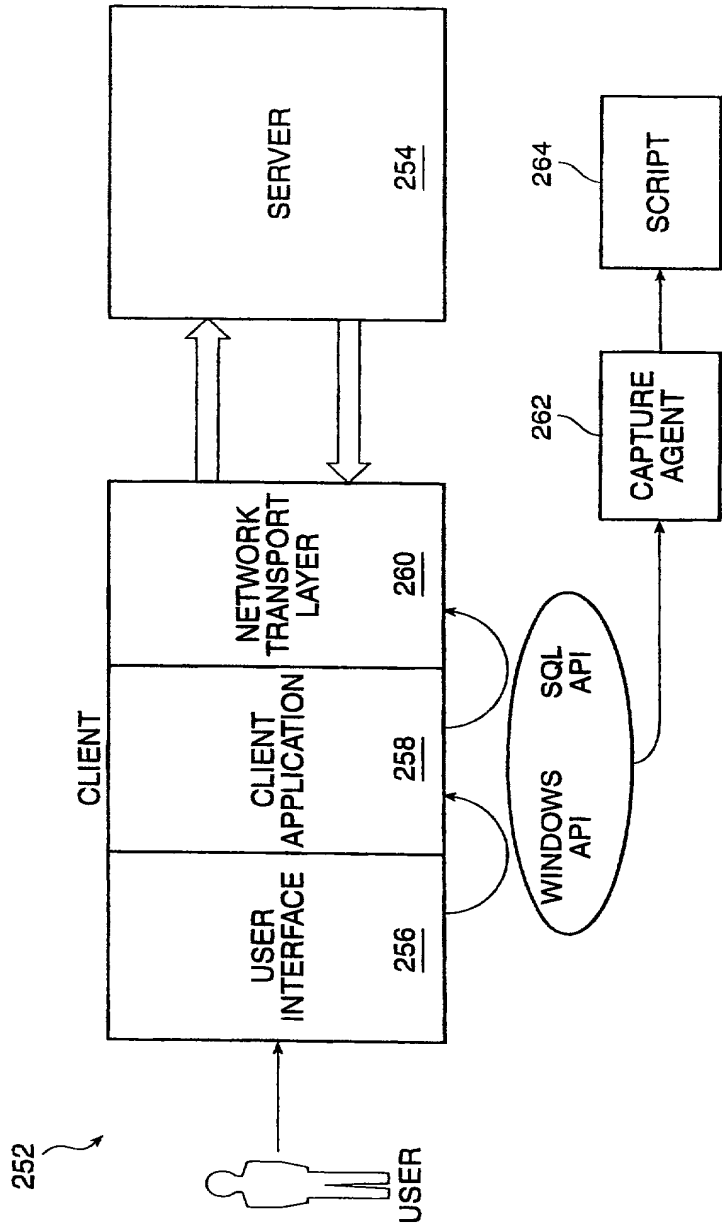


FIG. 5

5/13

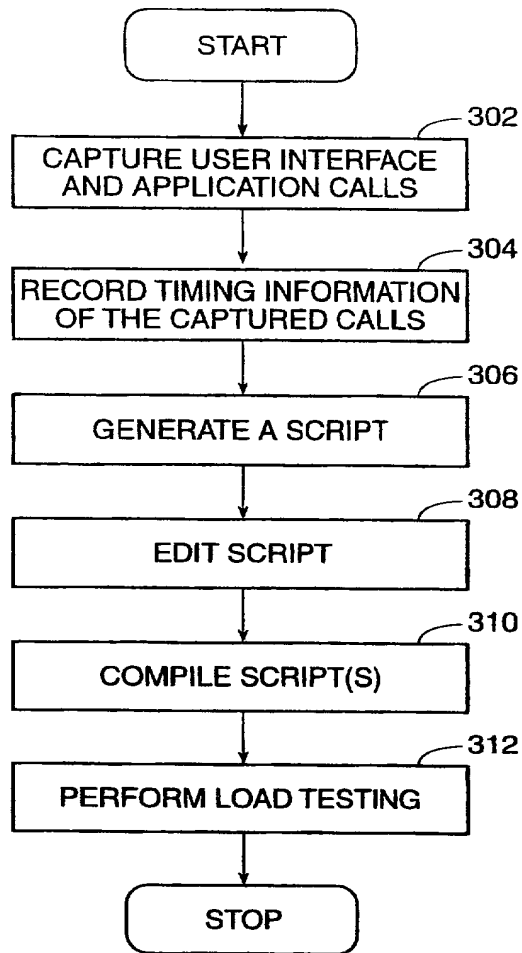


FIG. 6

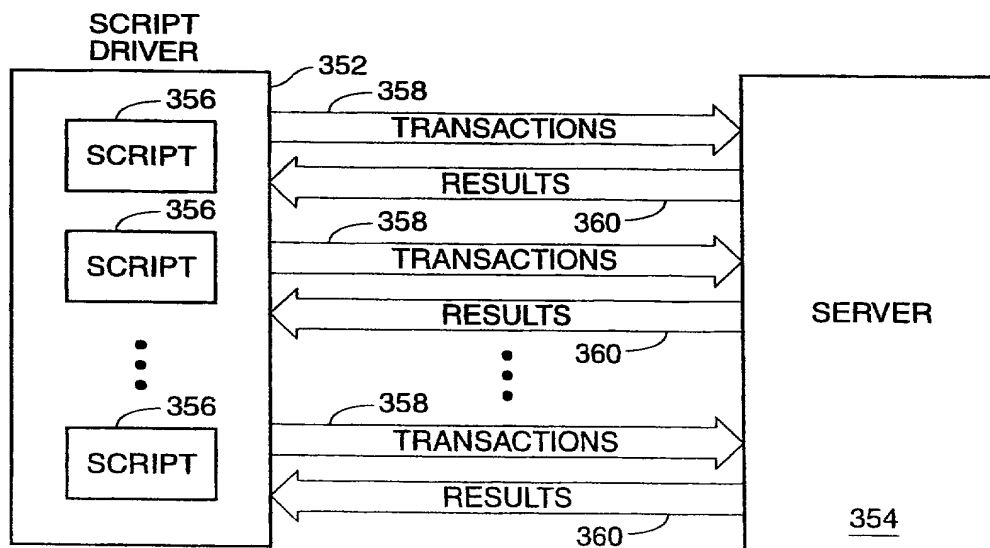


FIG. 7

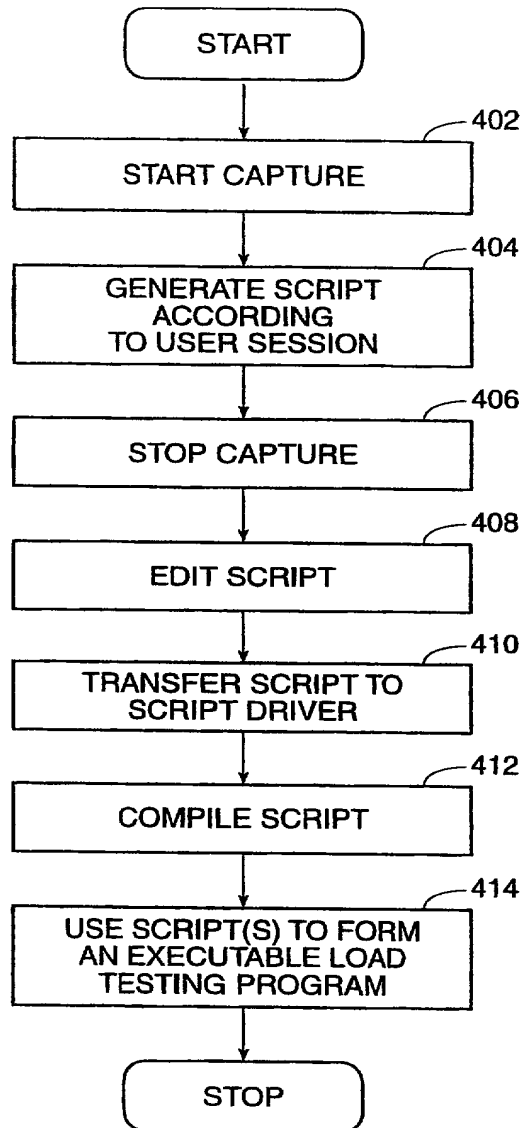


FIG. 8

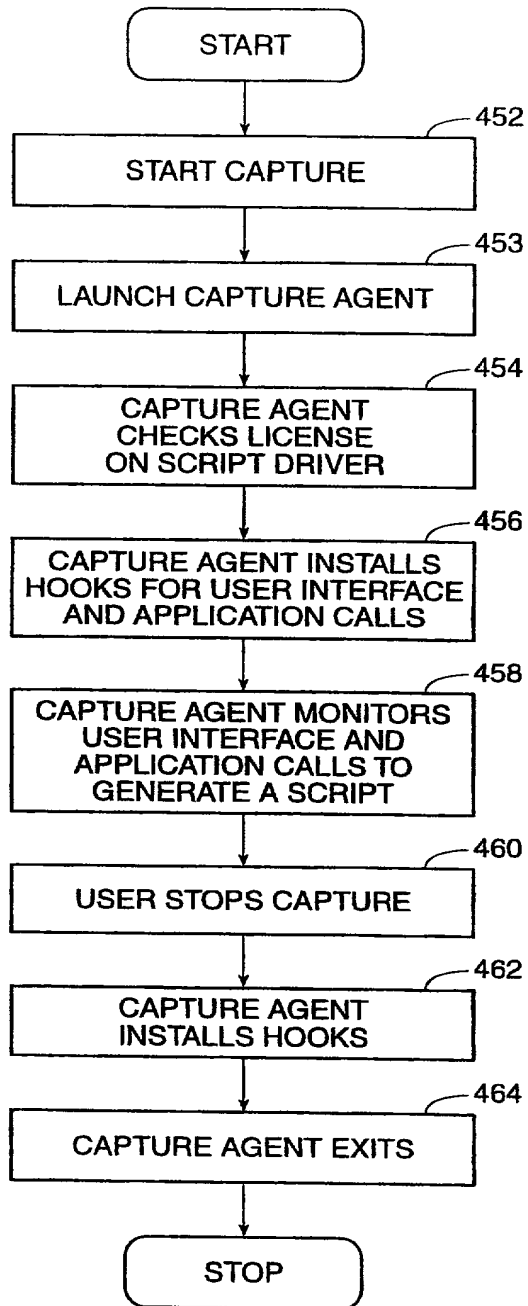
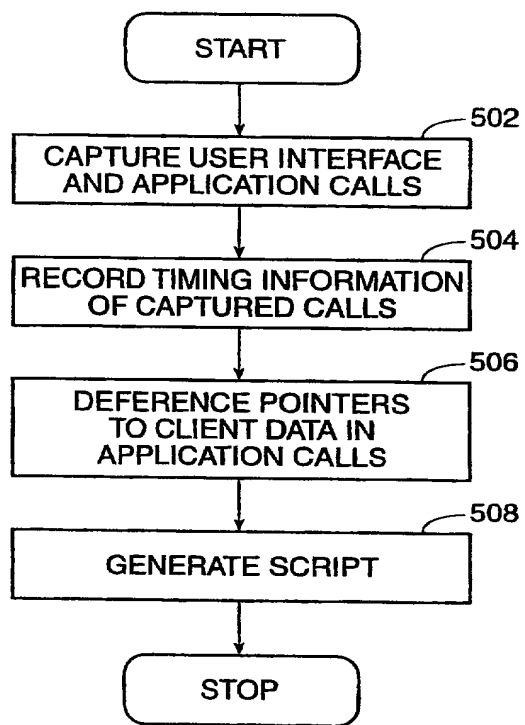


FIG. 9

*FIG. 10*

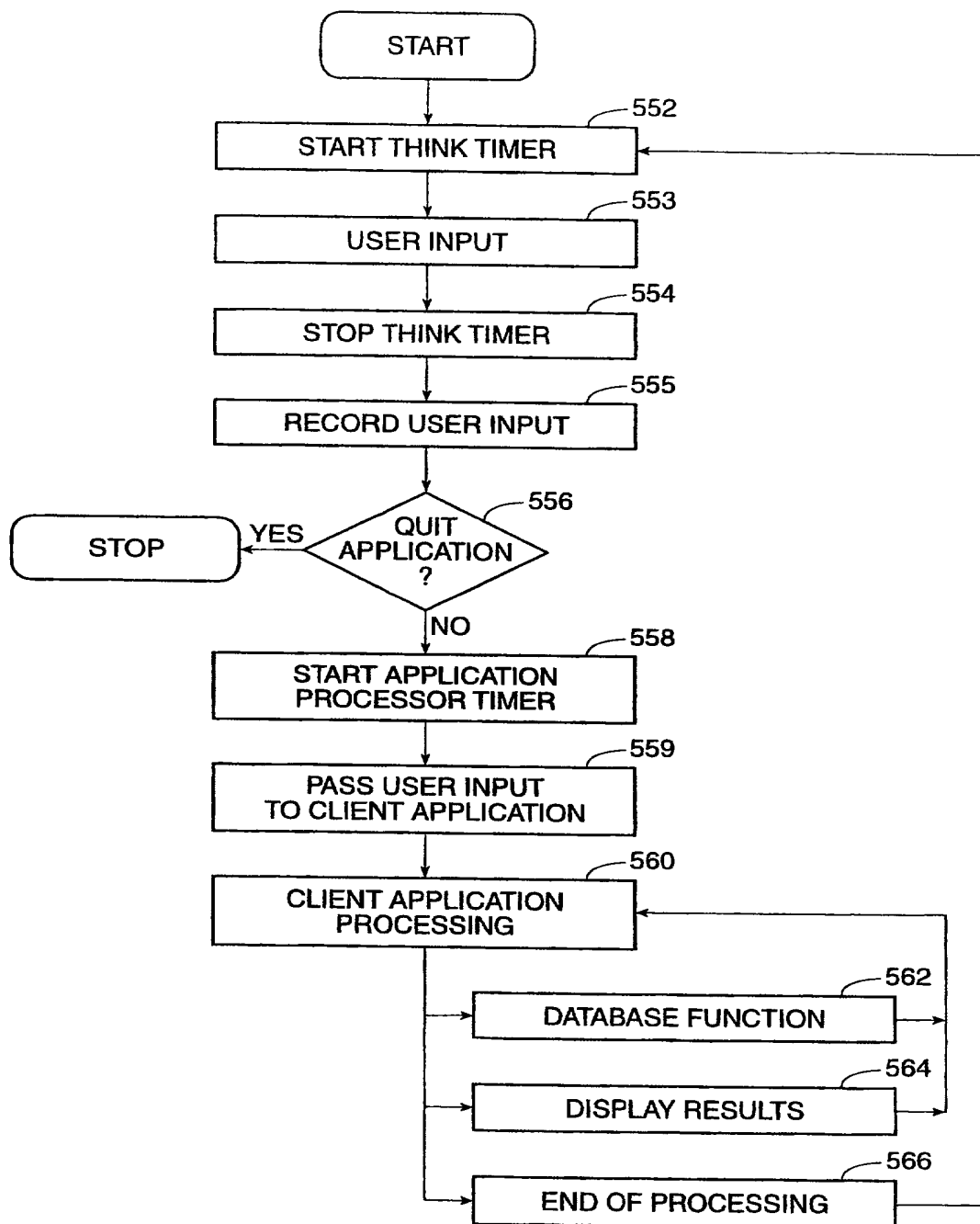


FIG. 11

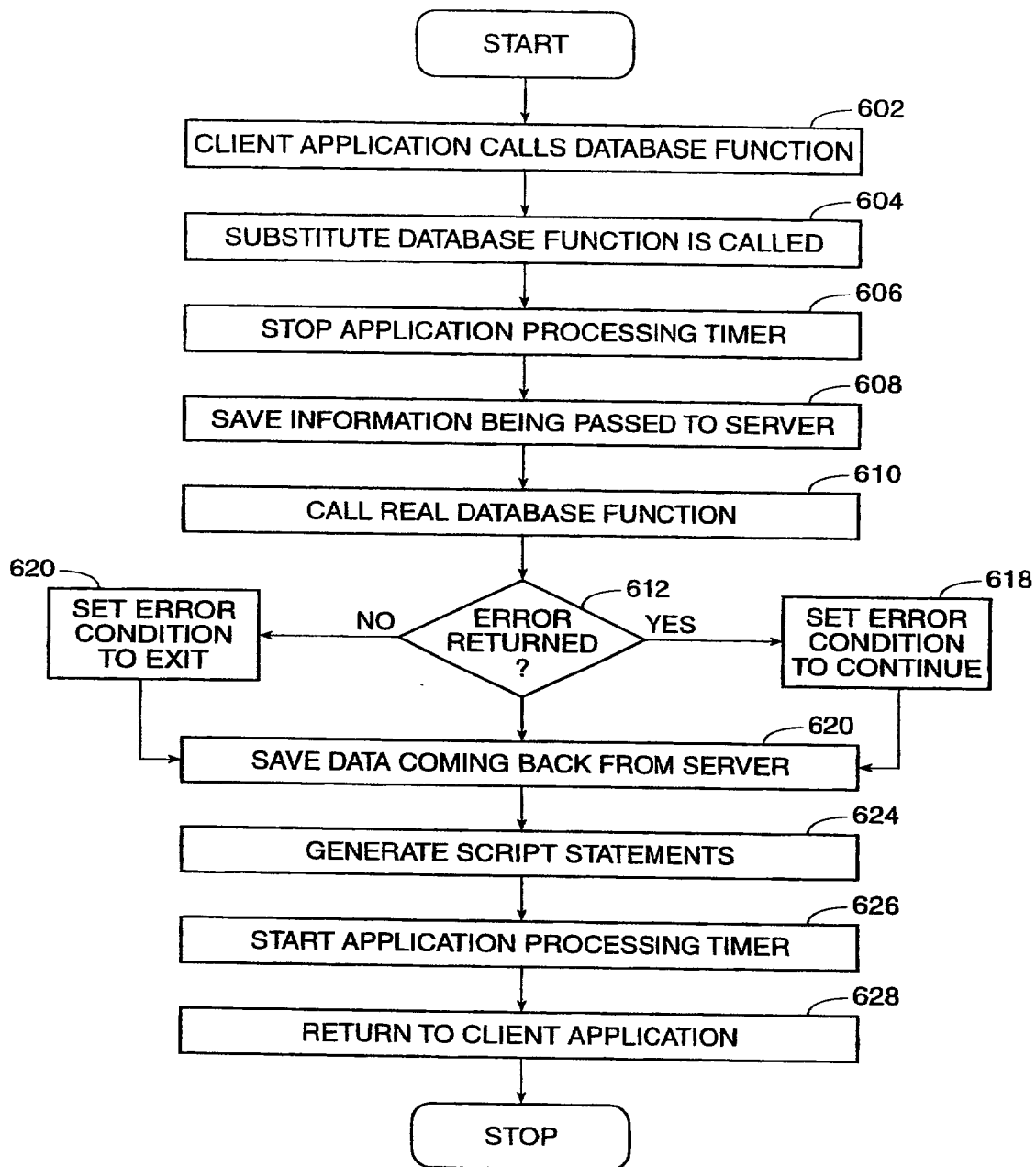


FIG. 12

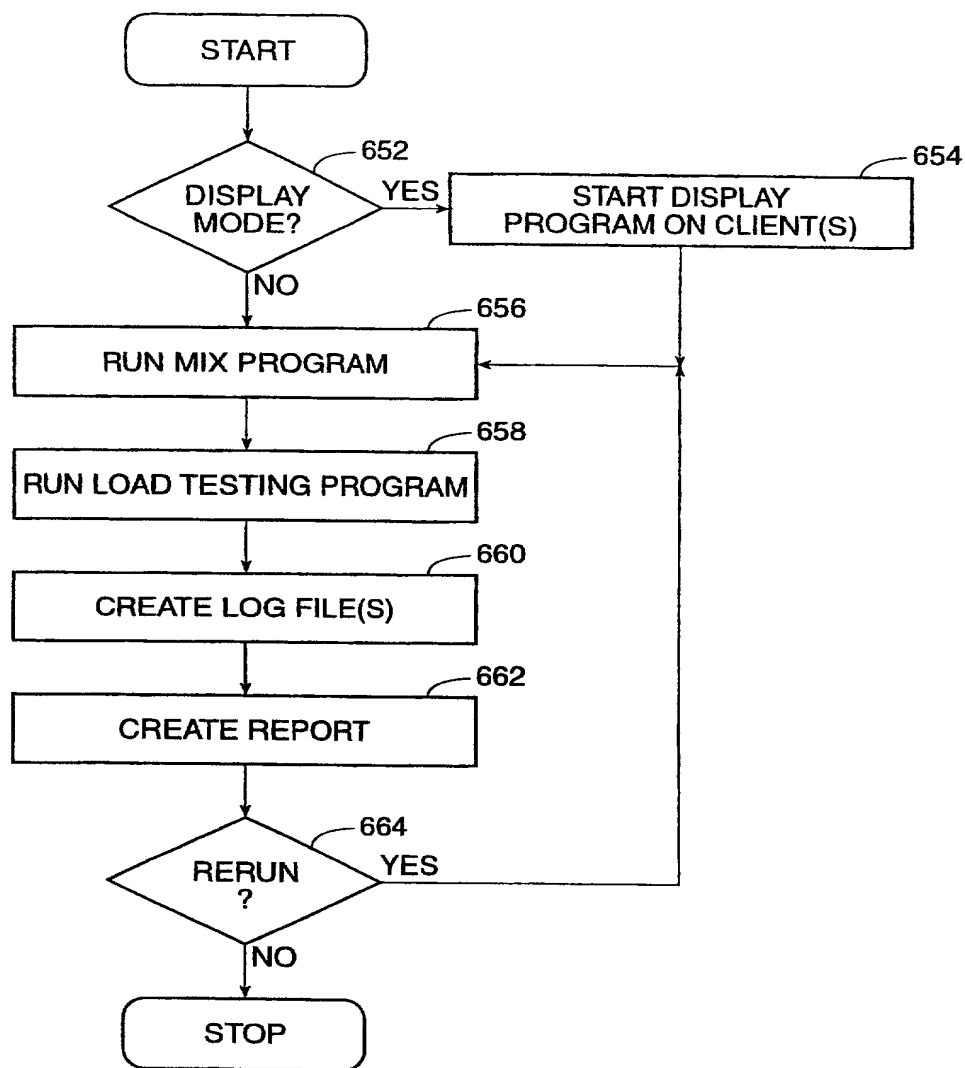


FIG. 13

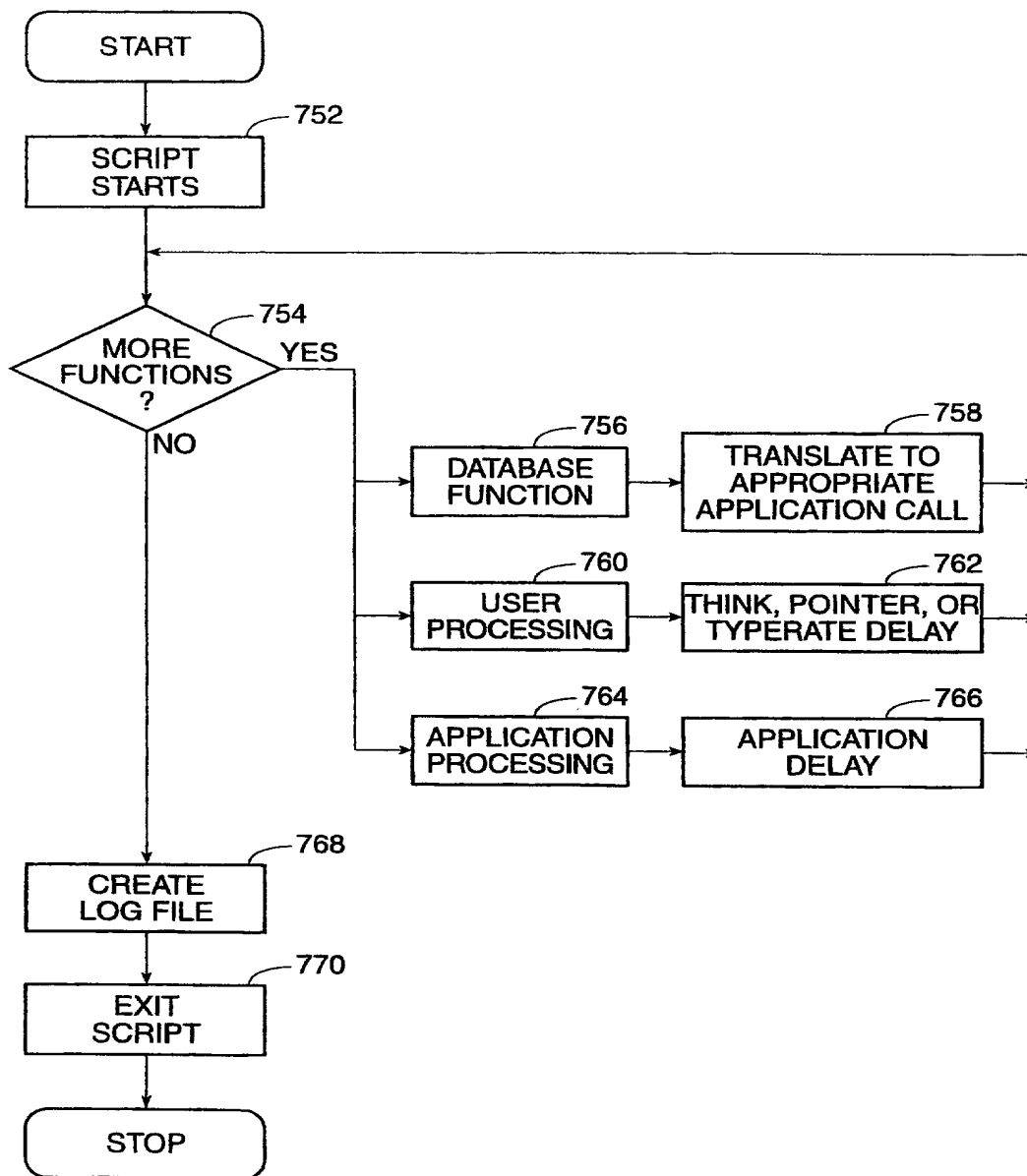


FIG. 15

DECLARATION AND POWER OF ATTORNEY

As a below named inventor, I declare that:

My residence, post office address and citizenship are as stated below next to my name; I believe I am an original, first and joint inventor of the subject matter which is claimed and for which a patent is sought on the invention entitled: **LOAD TEST SYSTEM AND METHOD**, the specification of which was filed December 22, 1995 as Application No. 08/577,278.

I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above. I acknowledge the duty to disclose information which is material to the examination of this application in accordance with Title 37, Code of Federal Regulations, Section 1.56. I claim foreign priority benefits under Title 35, United States Code, Section 119 of any foreign applications(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed.

Prior Foreign Application(s)

Country	Application No.	Date of Filing	Priority Claimed Under 35 USC 119
			Yes ____ No ____
			Yes ____ No ____

I claim the benefit under Title 35, United States Code, Section 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code, section 112, I acknowledge the duty to disclose material information as defined in Title 37, Code of Federal Regulations, section 1.56 which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

Application Serial No.	Date of Filing	Status
		____ Patented ____ Pending ____ Abandoned
		____ Patented ____ Pending ____ Abandoned

I hereby claim the benefit under Title 35, United States Code § 119(e) of any United States provisional application(s) listed below:

Application No.	Filing Date
60/007,590	November 24, 1995

POWER OF ATTORNEY:

As a named inventor, I hereby appoint the following attorney(s) and/or agent(s) to prosecute this application and transact all business in the Patent and Trademark Office connected therewith.

David N. Slone
Registration No. 28,572

Vernon A. Norviel
Registration No. 32,483

Michael J. Ritter
Registration No. 36,653

Eric H. Willgohe
Registration No. 34,755

James M. Heslin
Registration No. 29,541

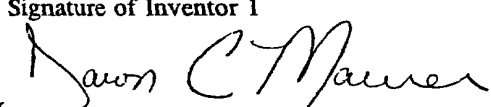

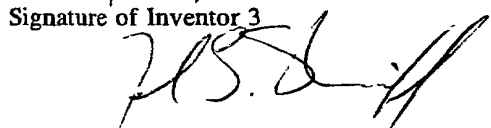
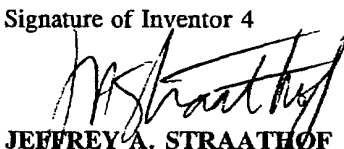
Paul C. Haughey
Registration No. 31,836

Send Correspondence to: Vern Norviel TOWNSEND and TOWNSEND and CREW Steuart Street Tower One Market Plaza, 20th Floor San Francisco, CA 94105	Direct Telephone Calls to: (Name, Reg. No., Telephone No.) Name: Michael J. Ritter Reg. No. 36,653 Telephone: (415) 326-2400
---	---

009207-46626550

Full Name of Inventor 1	Last Name MAURER	First Name DAWN	Middle Name or Initial C.	
Residence & Citizenship	City Centreville	State/Foreign Country Virginia	Country of Citizenship United States of America	
Post Office Address	Post Office Address 13702 Glassford Place	City Centreville	State/Country Virginia	Zip Code 22020
Full Name of Inventor 2	Last Name CHHINA	First Name RAMENDRA	Middle Name or Initial S.	
Residence & Citizenship	City Arlington	State/Foreign Country Virginia	Country of Citizenship United States of America	
Post Office Address	Post Office Address 2525 N. 10th Street, #618	City Arlington	State/Country Virginia	Zip Code 22201
Full Name of Inventor 3	Last Name SHERRIFF	First Name JOEL	Middle Name or Initial L.	
Residence & Citizenship	City Manassas	State/Foreign Country Virginia	Country of Citizenship United States of America	
Post Office Address	Post Office Address 10237 Bethany Court	City Manassas	State/Country Virginia	Zip Code 22110
Full Name of Inventor 4	Last Name STRAATHOF	First Name JEFFREY	Middle Name or Initial A.	
Residence & Citizenship	City Bethesda	State/Foreign Country Maryland	Country of Citizenship United States of America	
Post Office Address	Post Office Address 5200 Belvoir Drive	City Bethesda	State/Country Maryland	Zip Code 20816

I further declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issuing thereon.

Signature of Inventor 1  DAWN C. MAURER	Signature of Inventor 2  RAMENDRA S. CHHINA
Date 4/22/96	Date 4/22/96
Signature of Inventor 3  JOEL L. SHERRIFF	Signature of Inventor 4  JEFFREY A. STRAATHOF
Date 4-22-96	Date 4-22-96

APPLICATION FOR UNITED STATES PATENT
LOAD TEST SYSTEM AND METHOD

By Inventors:

Dawn C. Maurer
13702 Glassford Place
Centreville, Virginia 22020
A Citizen of United States

Ramenda S. Chinna
2525 N. 10th Street, #618
Arlington, Virginia 22201
A Citizen of the United States

Joel L. Sherriff
10237 Bethany Court
Manassas, Virginia 22110
A Citizen of the United States

Jeffrey A. Straathof
5200 Belvoir Drive
Bethesda, Maryland 20816
A Citizen of the United States

Assignee: Rational Software Corporation
2800 San Tomas Expressway
Santa Clara, CA 95051-0951

Entity: Large

Ritter, Van Pelt & Yi LLP
4906 El Camino Real, Suite 205
Los Altos, CA 94022
(650) 903-3500

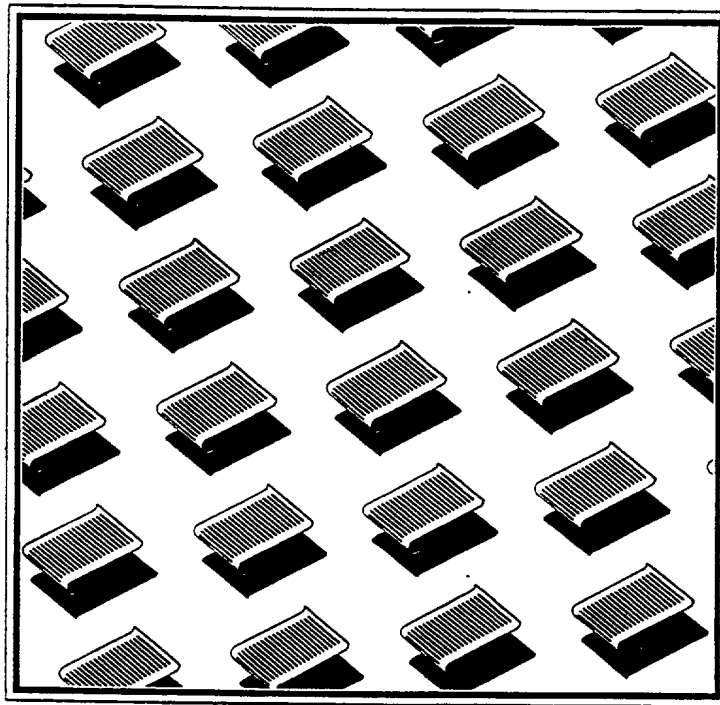
APPENDIX 1

© Copyright 1995
Pure Software, Inc.
All Rights Reserved

BOOK REVIEW

EMPOWER

Script Development



f o r E M P O W E R / C S



 **PERFORMIX**

09697994-103600

LIMITATION OF LIABILITY

PERFORMIX makes no warranty or representation of any kind, either expressed or implied, with respect to this software, updates, or documentation, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. This software, updates, and documentation are provided 'AS IS'. The entire risk as to their quality, performance, and results is assumed by you. Some states do not allow the exclusion of implied warranties, so the above limitation may not apply to you.

In no event will PERFORMIX be liable to you for any damages whatsoever (including but not limited to direct, indirect, special, incidental, or consequential damages) arising out of the use or inability to use the software, updates, or documentation even if PERFORMIX has been advised of the possibility of such damages. In particular, PERFORMIX is not responsible for any damages including but not limited to those resulting from lost profits or revenue, loss of use to the computer program, loss of data, claims by third parties, or for other similar damages. Some states do not allow the execution or limitation of liability for consequential or incidental damages, so the above limitation may not apply to you.

COPYRIGHT

The PERFORMIX documentation and the software are copyrighted and protected by both the Universal Copyright Convention and the Berne Convention. All rights are reserved. No part of this documentation nor the software may be copied, reproduced, translated, or transmitted in any form or by any means except as expressly described in your license agreement.

U.S. GOVERNMENT RESTRICTED RIGHTS

If this software and documentation are acquired by or on behalf of a unit or agency of the United States Government this provision applies. This software and documentation: (a) were developed at private expense, and no part of it was developed with government funds, (b) are a trade secret of PERFORMIX, Inc. for all purposes of the Freedom of Information Act, (c) are "commercial computer software" and "computer software documentation" subject to limited utilization as provided in the contract between the vendor and the government entity, and (d) in all respects are proprietary data belonging solely to PERFORMIX, Inc.

The enclosed software and documentation are provided with RESTRICTED AND LIMITED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR 52.227-14 (June 1987) Alternate III(g)(3)(June 1987), FAR 52.227-19(June 1987), or DFARS 252.227-7013 (c)(1)(ii)(October 1988), as applicable. Contractor/Manufacturer is PERFORMIX, Inc., 8200 Greensboro Drive, Suite 1475, McLean, VA 22102. Unpublished-rights reserved under the copyright laws of the United States.

EMPOWER/CS is a trademark of PERFORMIX, Inc. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Windows is a trademark of Microsoft Corporation. IBM is a registered trademark of International Business Machines Corporation. UNIX is a registered trademark of UNIX Systems Laboratories, Inc. Oracle is a registered trademark of Oracle Corporation. Sybase is a registered trademark of Sybase, Inc. All other trademarks are the property of their respective holders.

Software Version 1.0.1, User's Guide Version 1.0.1



PERFORMIX, Inc.
8200 Greensboro Drive, Suite 1475
McLean, VA 22102
(703) 448-6606 (phone)
(703) 893-1939 (fax)

Table of Contents

1.0 Welcome to EMPOWER/CS	1-1
1.1 Organization of the EMPOWER User's Guides	1-1
1.2 Organization of this Manual	1-2
1.3 User's Guide Conventions	1-3

2.0 Introduction to Load Testing with EMPOWER/CS.....	2-1
2.1 Why Use EMPOWER/CS?	2-1
2.1.1 Applications Development.....	2-1
2.1.2 Computer Purchase Decisions	2-2
2.1.3 Capacity Planning.....	2-2
2.1.4 Product Demonstrations.....	2-3
2.2 How Does EMPOWER/CS Work?	2-3
2.3 Testing Process	2-5
2.4 EMPOWER/CS Tools	2-6
2.4.1 Capture.....	2-7
2.4.2 Csccl.....	2-10
2.4.3 Mix.....	2-11
2.4.4 Reports.....	2-12
2.4.5 Draw.....	2-12
2.4.6 Monitor	2-14
2.4.7 Global Variables.....	2-14

3.0 Installation.....	3-1
3.1 The EMPOWER/CS Environment	3-2
3.2 Installing the User PC, the UNIX Script Driver, and the Database Server.....	3-2

3.3	Installing the EMPOWER/CS Software	3-2
3.3.1	Installing EMPOWER/CS on the UNIX Script Driver	3-3
3.3.2	Configuring the UNIX Driver for EMPOWER/CS.....	3-5
3.3.3	Installing EMPOWER/CS on the PC.....	3-6
3.4	Before Starting Capture	3-7

4.0	Capture.....	4-1
4.1	Suggestions for Executing Capture	4-3
4.2	Executing Capture	4-4
4.2.1	Options.....	4-7
4.2.1.1	Think Threshold	4-9
4.2.1.2	Bring to Front with	4-10
4.2.1.3	View Script.....	4-10
4.2.1.4	Database Traffic Only	4-11
4.2.1.5	Insert Timer	4-11
4.2.1.6	Database Chooser	4-12
4.2.1.7	Transfer Script after Capture	4-13
4.2.1.8	License Daemon	4-15
4.2.2	Directory	4-15
4.2.3	Capturing Application Activity into a Script File.....	4-16
4.2.4	Comments, Functions, and Timers.....	4-18
4.2.4.1	Inserting Timers	4-19
4.2.4.2	Inserting Functions	4-19
4.2.4.3	Comments	4-22
4.2.5	Completing Your Capture Session.....	4-22

5.0	Cscc.....	5-1
5.1	Cscc Syntax	5-1
5.2	Cscc Environment Variables	5-2
5.3	Compiling with Cscc	5-4
5.3.1	Specifying the Binary Name with -o.....	5-4
5.3.2	Preprocessing the Source Script with -E.....	5-5
5.3.3	Modular Script Design.....	5-6
5.3.4	Extending Modular Script Compilation with -F.....	5-10

5.3.5 Automatic Creation of Function Archives.....	5-11
5.3.6 Optimizing and Stripping the Script Binary.....	5-12
5.3.7 Excluding Help Information from the Script Binary.....	5-12
5.3.8 Excluding Monitor Code.....	5-13
5.3.9 The Compiler Command Line.....	5-13
5.4 Csc compilation Messages	5-14

6.0 Script Execution	6-1
6.1 Non-Display Mode	6-1
6.2 Script Execution in Display Mode.....	6-2
6.3 Script Execution Options.....	6-5
6.3.1 -d.....	6-5
6.3.2 Changing the Script ID.....	6-5
6.3.3 Specifying a Log File.....	6-6
6.3.4 Specifying Arguments.....	6-7
6.4 The Log File.....	6-8
6.5 What Comes Next?.....	6-12

7.0 Script Content and Enhancement.....	7-1
7.1 Script Content	7-1
7.1.1 General Script Functions.....	7-4
7.1.1.1 Begin/End Functions	7-4
7.1.1.2 Typerate and Pointerrate.....	7-5
7.1.1.3 Think Time Functions	7-7
7.1.1.4 Timeouts and Database Errors.....	7-9
7.1.1.5 Set, Unset	7-11
7.1.1.6 Mouse Activity	7-12
7.1.1.7 Keyboard Activity	7-14
7.1.1.8 Type.....	7-17
7.1.1.9 ButtonPush	7-19
7.1.1.10 InitialWindow	7-20
7.1.1.11 AppWait Delay	7-22
7.1.1.12 WindowRcv.....	7-23
7.1.1.13 CurrentWindow	7-24

EMPOWER/CS-V1.0.1

7.5.2.9	Fioreadchar.....	7-62
7.5.2.10	Fiowritechar.....	7-62
7.5.2.11	Fioskipline.....	7-62
7.5.2.12	Fioskipfield.....	7-63
7.5.2.13	Fioskipchar.....	7-63
7.5.2.14	Fioseek.....	7-63
7.5.2.15	Fiorewind.....	7-64
7.5.2.16	Fioautorewind.....	7-64
7.5.2.17	File Input/Ouput Function Examples.....	7-64
7.5.3	Pacing Functions.....	7-66
7.5.4	Advanced Use of Script Variables.....	7-68
7.5.5	Looping.....	7-71
7.5.5.1	Looping with the For Statement.....	7-71
7.5.5.2	Looping with the While Statement.....	7-72

8.0	EMPOWER/CS Tools	8-1
8.1	XY	8-1
8.2	Modules.....	8-2
8.3	Transfer.....	8-3
8.4	Tree	8-6
8.4.1	The Tree Structure.....	8-6
8.4.2	Using the Tree Tool.....	8-8

Index

1.0 Welcome to EMPOWER/CS

Welcome to EMPOWER/CS, PERFORMIX Inc.'s full-featured solution for efficient client/server load testing. With EMPOWER/CS, you can emulate multiple users from a single load testing machine to accurately measure response time and throughput of your client/server environment. EMPOWER/CS captures and replays actual client/server activities running from Microsoft Windows PCs. It stress tests applications and the database server by emulating multiple PCs that interact with the server and then measures client/server performance by summarizing response times.

Until now, client/server load testing required assembling individual PCs for each system-supported user, which could involve up to *hundreds* or *thousands* of PCs. With EMPOWER/CS, this costly and cumbersome process is reduced to **one** UNIX driver machine that emulates as many PCs needed to test your entire client/server environment.

You can use EMPOWER/CS to load test your client/server applications during the entire cycle of development, to determine your database server's performance under peak load conditions, and to help determine future needs of your client/server environment.

This User's Guide is designed to help you understand the testing processes and capabilities of EMPOWER/CS and to guide you through efficient client/server load testing.

1.1 Organization of the EMPOWER User's Guides

The complete documentation for EMPOWER/CS includes three user's manuals which contain general use information, installation instructions, technical reference material, and examples. The following list identifies each user manual:

EMPOWER/CS Script Development

Describes how to create and execute scripts to perform realistic load tests on your client/server environment. This process involves using the EMPOWER/CS tools, Capture and Csccl, and editing and enhancing your scripts

Multi-User Testing

Describes how to use the multi-user tools Mix, Extract, Report, Draw, Monitor, and Global Variables (GV) for emulating realistic loads and measuring performance

Reference

Describes commands, functions, and possible error messages for EMPOWER/CS. This manual also includes information for contacting PERFORMIX technical support

1.2 Organization of this Manual

EMPOWER/CS Script Development is divided into the following sections:

- | | |
|------------|---|
| Section 1: | Welcomes you to EMPOWER/CS |
| Section 2: | Introduces you to EMPOWER/CS and load testing for client/server environments |
| Section 3: | Outlines the procedure for installing EMPOWER/CS and setting up the testing environment |
| Section 4: | Describes the EMPOWER/CS tool, Capture, which captures user and PC activities to build executable scripts |
| Section 5: | Describes the EMPOWER/CS tool, Csccl, which compiles your scripts into an executable format |
| Section 6: | Describes the processes for executing scripts |
| Section 7: | Describes script content and methods for editing your scripts including step-by-step examples |
| Section 8: | Describes the EMPOWER/CS Windows tools |

1.3 User's Guide Conventions

The conventions followed in this User Guide are listed below:

Regular Font	Used for all regular body text
Arial Font	Represents elements of the MS Windows environment such as pushbuttons, window titles, user entries, etc.
Mono-spaced Font	Used for all command, function, and file names; for all examples; and, generally, for any computer generated text
Bold mono-spaced font	In examples, represents entries made by the EMPOWER/CS user
<code>[-S scriptid]</code>	In command syntaxes, text within these square brackets represents optional command parameters
<code>()</code>	Vertical lines separate command parameter choices
<code>...</code>	Within scripts, the ellipsis marks indicate that some script content was left out for brevity
<code>Endfunction()</code>	Parentheses are included with script functions mentioned in regular body text. For most functions, one or more parameters will be listed in the parentheses when the function is used in a script
<code>Parse()</code>	EMPOWER/CS script functions use initial capitalization
<code>csc</code>	EMPOWER/CS command names use all lowercase letters
Capture	When an EMPOWER/CS tool is mentioned within regular body text, it is shown in regular font with initial caps

The term, "SUT," refers to your client/server system under test. The client/server SUT includes the database server and the database(s) on that server used for your emulation.

EMPOWER/CS tests the performance of a client/server application or system by emulating application activity of multiple PC users and by gathering response time data. Using EMPOWER/CS eliminates the need for many PCs to stress test your client/server system under test (SUT) because you can emulate multiple PC users on a single UNIX driver machine applying a realistic load to the SUT. EMPOWER/CS tools allow you to capture actual user activity in a script file, combine and execute several scripts at once, and collect performance statistics.

You can use EMPOWER/CS to support applications development, computer purchase decisions, capacity planning, and product demonstrations.

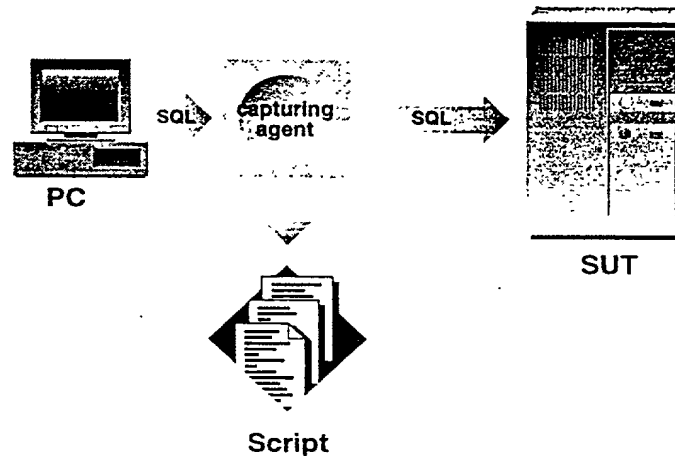
After you develop an application, most problems do not arise until your customer tries to use the application during peak load conditions. Load testing with EMPOWER/CS measures the true scalability and performance of new client/server applications and systems before they are introduced to actual users. By emulating multiple users with EMPOWER/CS, you can identify problems and defects from the very beginning of the development cycle up to new releases of the product.

Companies that do not load test their client/server products risk introducing applications to the marketplace that fail to meet critical user expectations.

2.2 How Does EMPOWER/CS Work?

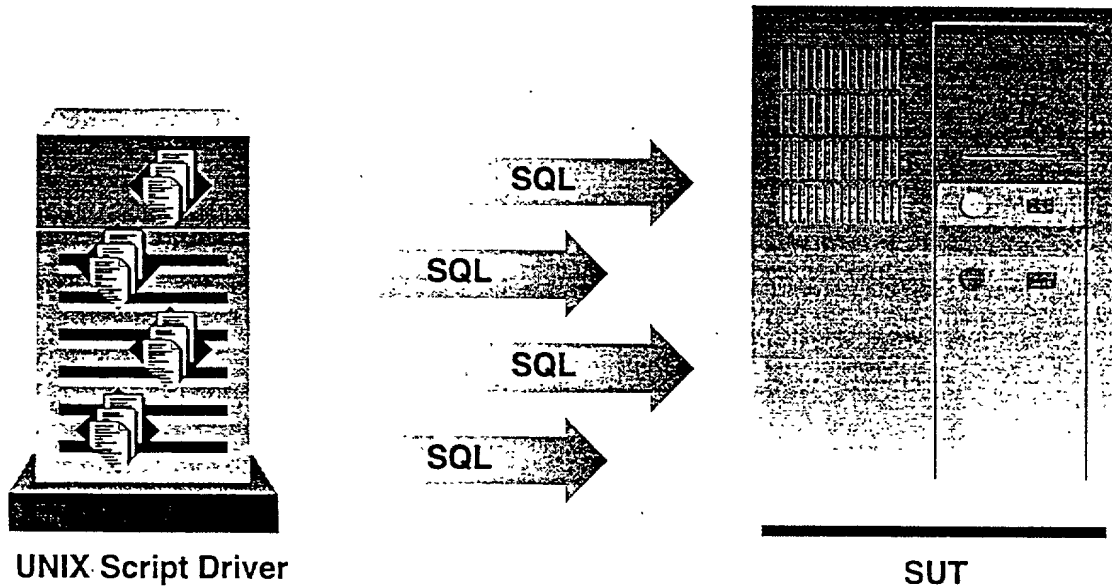
EMPOWER/CS is highly effective in demonstrating your client/server product or application to potential customers. A performance test emulating actual user and database activity provides your customers with a feeling of confidence that your client/server application will perform in the field as promised. This is especially useful if you do not have reference sites for your potential customers' workload.

EMPOWER/CS places a capturing agent on a Microsoft Windows PC that records dialog between the PC and the SUT. From this interaction, the capturing agent builds a script that is suitable for executing multi-user tests:

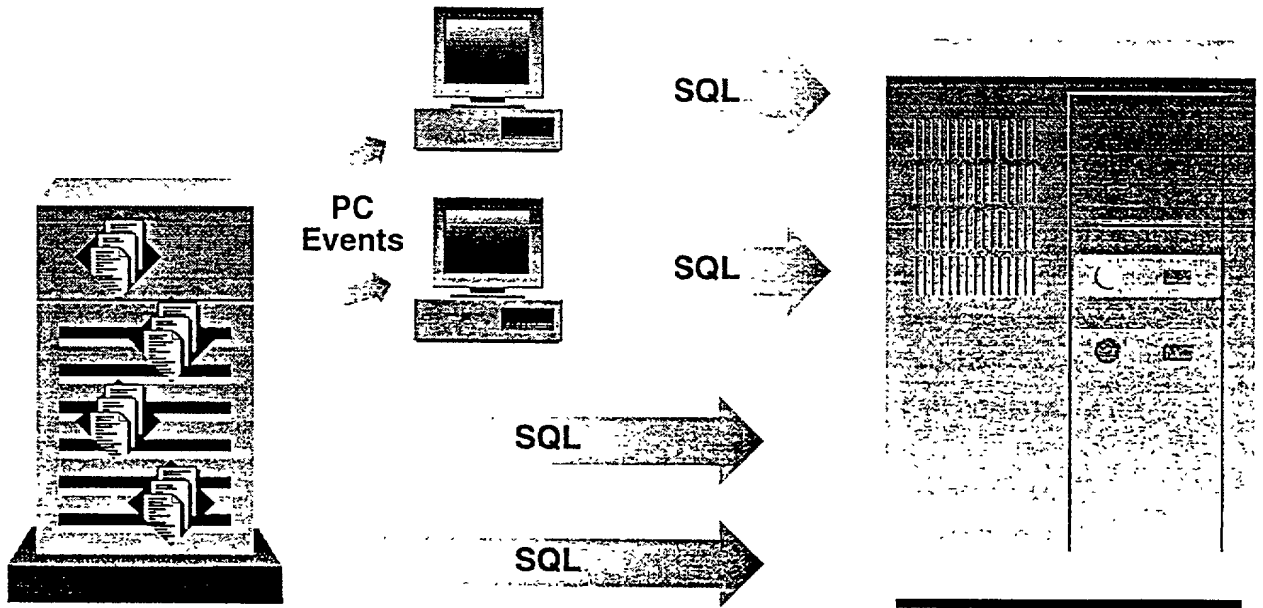


Because EMPOWER/CS requires the multi-tasking features of UNIX to run multiple scripts, scripts are transferred to a UNIX driver machine where they are compiled and executed. With EMPOWER/CS, the UNIX driver can combine and execute several scripts simultaneously, and the scripts can be edited and enhanced to emulate more realistic client/server activity. Only the UNIX script driver and your SUT are required for the emulation.

When the script is executed on the UNIX script driver, it acts as a client to interact with the SUT emulating captured PC activities. As the script executes, the SUT assumes it is servicing SQL requests from an actual PC, thus, creating a true multi-user test environment:



You need a PC only for capturing user activity and, optionally, for Display mode which displays emulated activity on screen as a script executes:



2.3 Testing Process

The first step for load testing your client server environment with EMPOWER/CS is capturing PC and SUT interactions into a script file. When capturing scripts, changes to your client/server application are not required. The capturing tool only requires that you perform the most common user activities. To build complete scripts, EMPOWER/CS records mouse and keyboard strokes, SQL requests to the SUT, and data returned to the PC. The resulting scripts are transferred to the UNIX script driver.

Before executing scripts, you must compile them on the UNIX driver machine. The executing script binary replaces the PC to interact with the SUT.

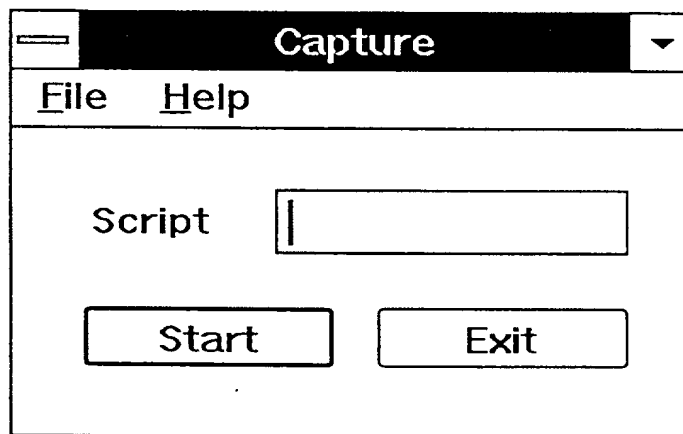
To produce the most accurate test results, you can edit your scripts to make them unique to your environment. Because EMPOWER/CS scripts are actually C language programs, they can be enhanced or supplemented as required. You can vary scripts by branching, altering data, and inserting loops. You also can change values entered in queries and updates during Capture (because in reality, all users would not query and update identical records continuously). User entries can be substituted by generating random values, sharing pools of input on the driver, accessing data returned from the SUT, or passing data between the scripts.

While tests are running, you can monitor scripts to verify progress, debug running scripts, and examine current response times. After your tests execute, you can generate reports that summarize interactive response time and peak throughput supported by your system. Valuable performance information such as response time averages, minimums, maximums, percentiles, transaction categories, and client processing times can be assembled within minutes.

EMPOWER/CS includes and uses the following tools: Capture, Csc, Mix, Extract, Report, Draw, Monitor, and Global Variables.

[illegible]

Capture is executed from Microsoft Windows on your PC to capture actual PC and SUT interactions in a script file. You simply activate the Capture icon, perform the user activity you wish to test, and then transfer your captured script file to the UNIX driver. Many options are available when capturing your script such as inserting user think times into the script file, adding comments and functions to the script, and automatically transferring scripts to the UNIX driver. The following example depicts the Capture command window.



A sample EMPOWER/CS script file, `script1.c`, follows. Such user and database activity as mouse events, logging on to the database, and processing a query were captured into this script (Some script content was left out for brevity):

(continued on following page...)

(continued on following page...)


```

Dbset(CUR1, FETCHSIZE, 64);
Fetch(CUR1);
while (GetNextRow(CUR1) != NOMOREROWS);

Endtimer(Accounts_Payable);

...

Begintimer("Accounts");

Commit(LOG1);

WindowRcv("PtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");

Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE);

Endtimer("Accounts");

Endscenario("script1");

```

2.4.2 Cscc

After the script file has been transferred to the UNIX driver, you should compile the script into a machine language version. The Csccl tool compiles the script which can be executed at the command line. Because the script is a C language program, additional C language statements can be added to enhance the script. In the following example, we will compile the script file `example1.c` into an executable binary:

```
$ cscd script1
```

The script now can be executed from the command line:

```
$ script1
```

Mix emulates an actual multi-user environment by executing multiple scripts after they are compiled by Csc. A script table and, optionally, a command file specify the number of users to emulate, the name of the script or scripts each user will execute, delay times between the start of each script, and the termination condition of the test. Each script that executes produces a log file used for preparing statistical reports.

In the following examples, suppose you have created four scripts. Your mix table, `table1`, could look like the following:

```
user1, query log1
user2, update log2
user3, accounts log3
user4, customer log4
```

With Mix, these scripts can be executed in a multi-user test:

```
$ mix
Mix: EMPOWER/CS V1.0.0, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95

mix> use table1
mix> start all
[user1] started
[user2] started
[user3] started
[user4] started
mix> user1 terminated (3/4) running
user2 terminated (2/4) running
user3 terminated (1/4) running
user4 terminated (0/4) running

mix> quit
$
```


Figure 1

Generating performance reports from one or more executed scripts is accomplished in two steps. First, you must use the Extract tool which parses each script's log file and records response time information in a set of flat ASCII files.

```
$ extract log1.1 log2.1 log3.1 log4.1
```

The Report tool then uses these ASCII files to produce statistical reports that describe response time and system throughput. (*Note:* You also may generate reports from your own statistical or graphics programs.) Report will generate an EMPOWER/CS standard report similar to the following:

```

$ report

EMPOWER Standard Report

Date:          Fri Jan 27 14:49:14 1995
Start time:    14:42:42
Stop time:     14:43:23
Duration:      00:00:41
Mix:           4 users
Unit:          seconds

Scenario      Total  Finish Thruput  Median Average Minimum Maximum Std-Dev
-----
query         1      1      0.02   31.26   31.26   31.26   31.26   0.00
update        1      1      0.02   22.07   22.07   22.07   22.07   0.00
accounts      1      1      0.02   30.95   30.95   30.95   30.95   0.00
customer      1      1      0.02   22.18   22.18   22.18   22.18   0.00
Overall       4      4      0.10   26.56   26.62   22.07   31.26   4.49
-----
Function      Total  Finish Thruput  Median Average Minimum Maximum Std-Dev
-----
logout1       2      2      0.05   4.35    4.35    4.33    4.36    0.01
-----

```

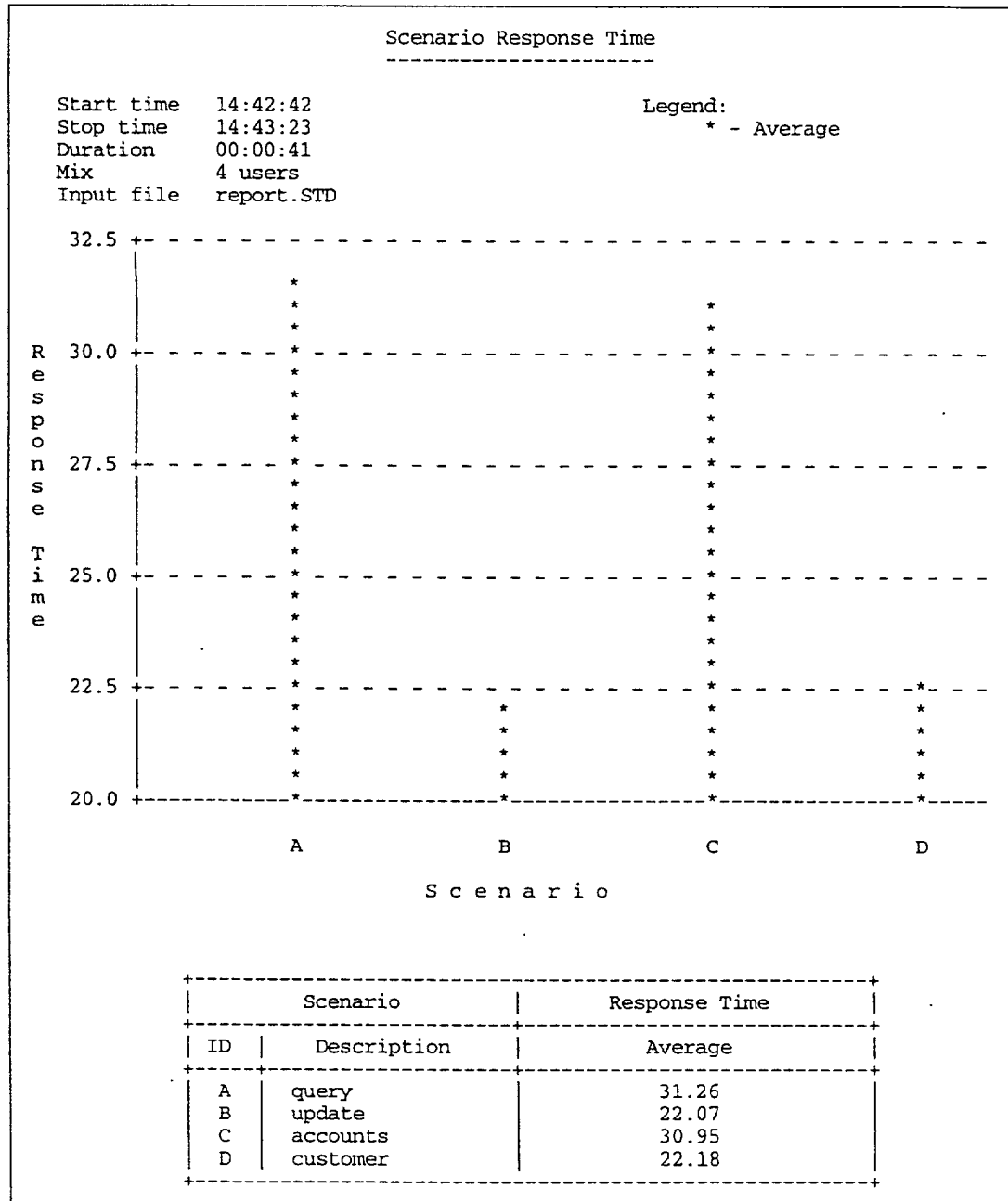
```
Date:          Fri Jan 27 14:49:14 1995
Start time:    14:42:42
Stop time:     14:43:23
Duration:      00:00:41
Mix:           4 users
Unit:          seconds
```

Function	Total	Finish	Thruput	Median	Average	Minimum	Maximum	Std-Dev
logout1	2	2	0.05	4.35	4.35	4.33	4.36	0.01

[illegible]

Draw accepts one or more EMPOWER/CS reports to produce bar charts that depict relationships among performance results. These relationships can summarize a

single multi-user test or a series of multi-user tests in which each test contained a different user level or system configuration. The following example is a typical EMPOWER/CS bar chart:




```

Fri Jan 27 14:42:58      EMPOWER/CS MONITOR V1.0      (c) PERFORMIX, Inc. 1995
View 1 of 7  Script 1 of 4      Running      ScriptId Sorting ^  Interval 5

ScriptId  __Script  State  _____  LastEvent  _____  CurrentWindow
user001   query    bpush  ><ButtonPush><ButtonPush><ButtonPush>  Employee Records
user002   update   appwt  ButtonPress>c:\acct\acct^M<ButtonPush>  Acct Application
user003   accounts appwt  nPress><LeftButtonPress>c:\acct\acct^M  Accounting
user004   customer type  <LeftButtonPress><LeftButtonPress>c:\  Run

```

3.0 Installation

EMPOWER/CS was designed to operate in an environment in which the SUT can not distinguish between actual PC users and the UNIX script driver. The PC, UNIX script driver, and SUT must be connected along the same communication network. The EMPOWER/CS software does not affect communication between the user PC and the SUT as it captures their activity.

Before a Capture session begins, the PC will run a licensing check with the UNIX driver to verify execution of Capture. If the PC can not communicate with the UNIX machine, you will not be able to execute Capture. The PC must also be linked to the UNIX script driver so that it may drive the PC during script execution in Display mode.

If the PC can not communicate with both the UNIX driver and the SUT, then EMPOWER/CS can not capture client/server activity. If the UNIX driver and the SUT can not communicate, then the UNIX driver can not execute a multi-user emulation.

The Capture, Display, and Tools elements of EMPOWER/CS run on a Microsoft Windows PC and the remaining EMPOWER/CS software runs on the UNIX script driver. So that the PC can communicate with the UNIX script driver for the licensing check and for script execution in Display mode, the PC must have installed the dynamic link library `winsock.dll` Version 1.1. The PC application communicates with the database according to how you have set up your client/server environment. Both the PC and the UNIX driver must have the appropriate database libraries installed to communicate with the database on the server.

When a script is executed, the PC no longer is required because the UNIX driver replaces the PC user to interact with the SUT. However, if you choose to observe a script as it progresses, you can activate the "Display" option for the script to display captured user activity on the PC.

3.1 The EMPOWER/CS Environment

The following equipment is required to capture user activity with EMPOWER/CS:

- ☐ One IBM-compatible PC with keyboard, mouse, and monitor running Microsoft Windows software, winsock.dll Version 1.1, database libraries that assist with communicating with the SUT, and the EMPOWER/CS components Capture, Display, and Tools
- ☐ The UNIX script driver machine with database libraries that provide comparable communication with the SUT and EMPOWER/CS software
- ☐ The database server
- ☐ One communications network connecting the PC, the UNIX script driver, and the database server

3.2 Installing the User PC, the UNIX Script Driver, and the Database Server

You should follow the appropriate installation procedures described in the owner's manuals for the user PC, the UNIX script driver, and the database server.

3.3 Installing the EMPOWER/CS Software

You must complete two installations to fully operate EMPOWER/CS. The EMPOWER/CS components (Capture, Display, and Tools) must be installed on the PC and the remaining EMPOWER/CS software must be installed on the UNIX script driver.

Step 5: Use the `tar` command to read in the tape. Some examples of `tar` commands are listed below:

```
$ tar xov
$ tar xovf /dev/rSA/qtape1
```

Step 6: Change to the `Install` directory and execute the `eminstall` command. You will be instructed to contact PERFORMIX and provide your machine identification code to obtain your installation password, for example:

```
$ cd Install
$ ./emininstall
```

You will receive a message similar to the following:

Please print the "../Install/machineid" file, note your name, fax number, and phone number, and fax it to Performix at 703-893-1939.

We'll generate your installation password and fax it back as soon as possible. When you get your installation password, re-run `emininstall`.

If you are unable to send a fax, please call Performix at 703-448-6606.

This message is stored in the machineid file which can be printed and faxed to PERFORMIX. You will be given a special password consisting of all alphabetical characters, for example, JKLMNOPQRS-ABCDEFGHIJLMNOPWXYZ.


```
$ ./emininstall
Installation password:  JKLMNOPQRS-ABCDEFGHLMNOPWXYZ
installing ../lib/empowercs.a...
installing ../lib/empowercsm.a...
installing ../bin/csccl...
installing ../bin/draw...
installing ../bin/extract...
installing ../bin/gv_cmds...
installing ../bin/mix...
installing ../bin/mon...
installing ../bin/report...
```

```
$ ranlib lib/*.a
```

3.3.2 Configuring the UNIX Driver for EMPOWER/CS

Step 2: Define your shell environment variable `EMPOWER` to be the directory in which `EMPOWER/CS` was installed.

Step 1: Insert the Installation diskette into your disk drive.

Step 2: From the DOS prompt, switch to the appropriate disk drive. Execute the DOS install command from the correct disk drive by typing the name of the drive and `csinstal` (Example: `a:\csinstal`). If Microsoft Windows is already running, in the Program Manager window, choose **Run** from the File Menu. In this window, type the name of the drive and the install command (Example: `a:\csinstal`) and select **OK**.

Step 3: Follow the instructions that appear on screen for the remainder of the installation.

Before starting Capture, you must run `eld` on the UNIX machine. You should start `eld` as super-user (`root`) and you must set and export the `EMPOWER` environment variable to the local directory that contains the `EMPOWER/CS` software.

Example:

```
# EMPOWER=/usr/empower
# export EMPOWER
# eld
```

You now should be able to use the EMPOWER/CS software. Confirm the installation by starting Microsoft Windows on your PC and activating the EMPOWER/CS window and then the Capture icon.

The software installation and setup are now complete. Remove the EMPOWER/CS tape from the tape drive and store it, with the EMPOWER/CS diskette, in a secure location.

4.0 Capture

The EMPOWER/CS Capture tool is used to capture communications between a user PC and the SUT to build executable scripts. After you complete a Capture session, the assembled scripts can be executed from a UNIX script driver to emulate numerous users of the client/server system.

When activated, Capture records all interactions between the PC and SUT including mouse and keyboard strokes, SQL requests to the SUT, and data transmitted to the PC. Automatically, this information is stored in a script file with a ".c" extension in a specified directory on your PC to be used for subsequent script execution. The .c extension implies that the file is recorded in C language syntax for later C compilation and execution. Refer to Sections 5 and 6 of this manual for more information on script compilation and execution.

The following script segment illustrates the types of user functions and database traffic contained in a script. This script contains such captured activity as user entries, connection and logging on to the database, and execution of a query:

```
AppWait(0.22);
WindowRcv("SfAcSfDwDwAcSfSfSfPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");
WindowRcv("PtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");

CurrentWindow("Program Manager",36,26,545,333);

SysKeyPress(VK_MENU);

Type("fr");

AppWait(0.06);
WindowRcv("CoCwCwCwCwCwCwCwCwSfAcPt");

CurrentWindow("Run",72,77,450,238);

Type("c:\\acct\\acct^M");
WindowRcv("CwCwCwCwCwCwCw");

Openenv(ORACLE,VERSIONV6V7);
```

(continued on following page ...)

[illegible]

4.1 Suggestions for Executing Capture

If you plan to execute EMPOWER/CS scripts in Display mode, you must ensure that the steps captured in the application to be tested are repeatable, meaning that all captured mouse activity can be *precisely* repeated by the UNIX script driver machine during script execution. Capturing mouse clicks in each activated window is not recommended because the recorded mouse locations, i.e., xy coordinates on screen, may not apply during script execution. For instance, windows do not always open in the same location each time they are activated. During script execution in Display mode, the recorded xy coordinates in the mouse event functions could be invalid and cause the script to malfunction by not activating the correct locations to activate applications or commands. Maximizing windows will ensure that all mouse actions are repeatable because the window will cover the entire screen allowing recorded xy coordinates to remain the same in Display mode.

Since keyboard presses and commands are more precise, we recommend that you use the keyboard as much as possible to activate windows, commands, or applications. Some keyboard capabilities are listed below:

- Alt + Space Bar to open system menus
- Alt + F to open file menus
- Control + F4 to close program windows
- Arrow keys to move through menus or among icons

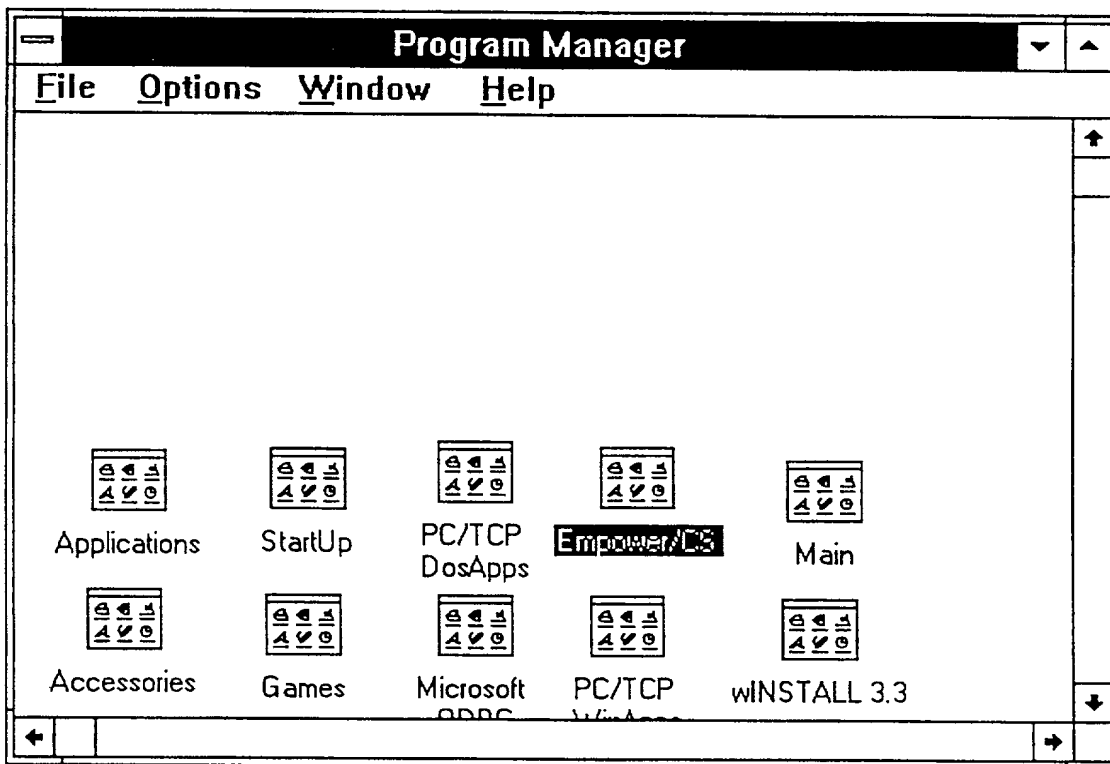
Refer to your MS Windows user guide for full instructions on using the keyboard to control your Windows desktop.

Capture is executed from Microsoft Windows on your PC.

If Windows is not currently running on your PC, from the DOS prompt, type "win" and press return.

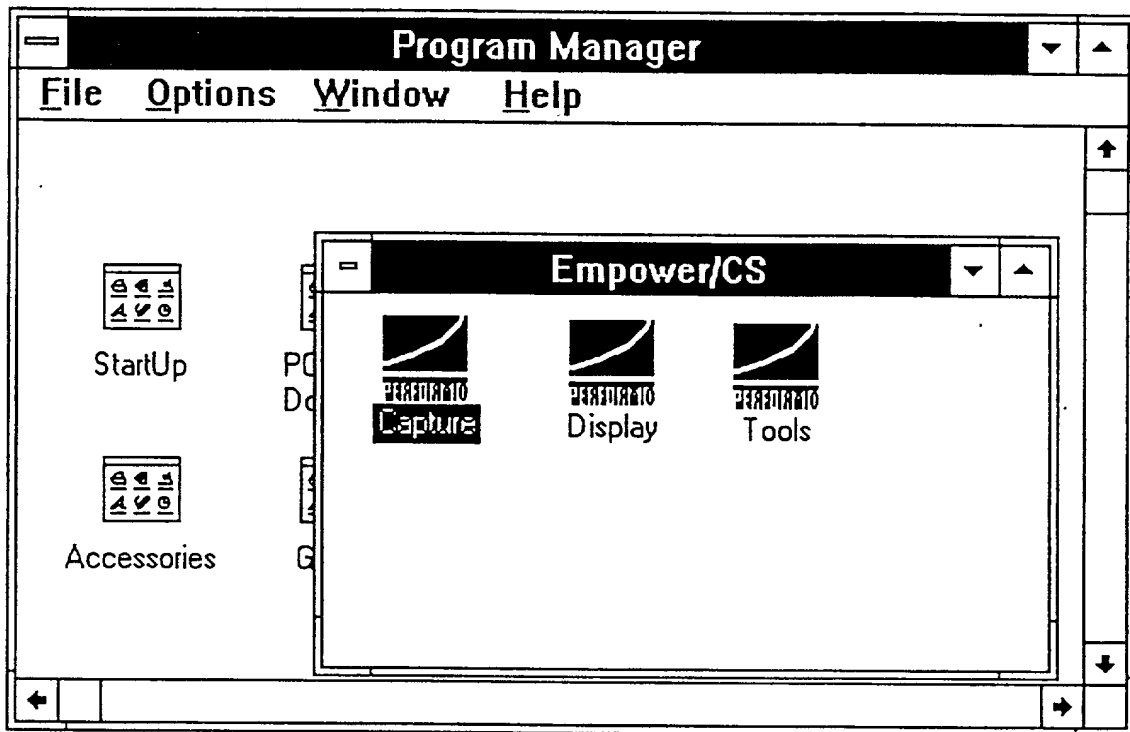
The Windows desktop and Program Manager window will open containing various program group icons. Activate the EMPOWER/CS program group.

Example:



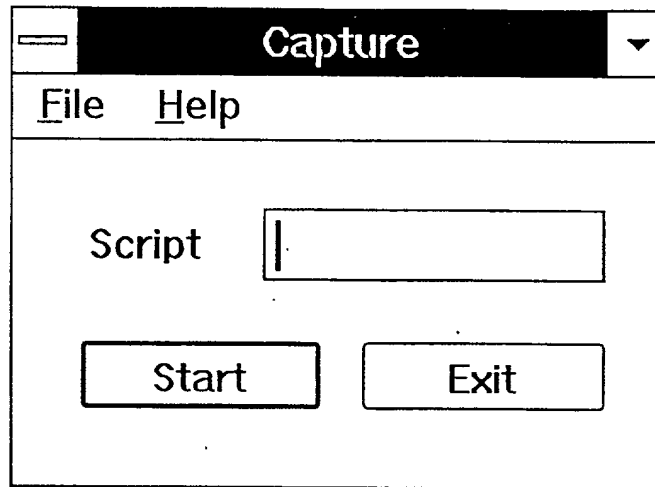
The following EMPOWER/CS window will open which includes three program-item icons: Capture, Display, and Tools.

Select the Capture icon:



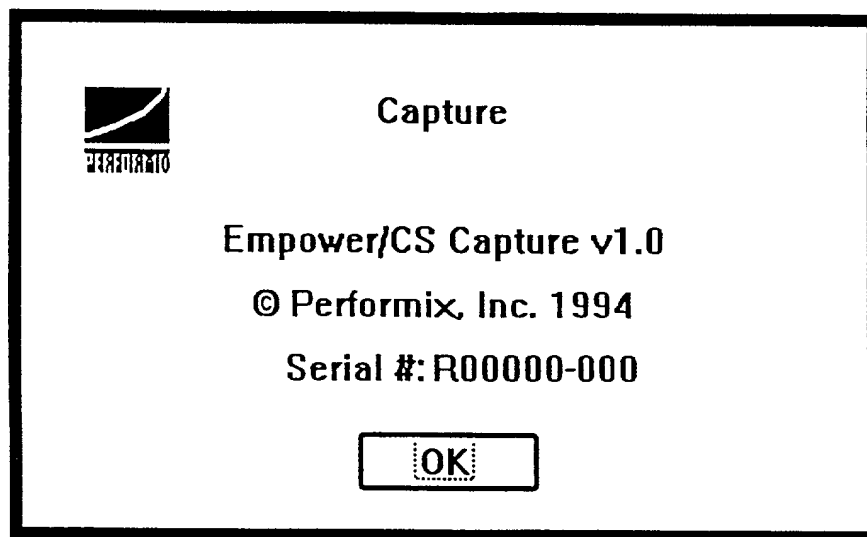
Before you may begin Capture, the PC must set up an initial communication or licensing check with the UNIX script driver. If this initial contact can not be made, you will not be able to execute Capture and will receive an error. *Note:* You must be sure to run `elc` as root on the UNIX machine before starting Capture on the PC. (Refer to Section 3.3.3 of this manual).

If the licensing check was successful, the following Capture command window will open prompting you to enter a script name:



This Capture window includes two items in the Menu Bar: File and Help. The File menu includes **O**ptions..., **D**irectory..., and **E**xit. The Help Menu (located in all the EMPOWER/CS windows) includes **A**bout... for listing EMPOWER/CS copyright information.

The About Capture screen is shown below:



Before you enter a script name to begin capturing, you may wish to change some of the Capture menu options. Under the File Menu, select Options....

4.2.1 Options

When you choose Options..., the following Options window will open:

Capture Options

Think Threshold ☐ Database traffic only

Bring to front with ☐ Insert timers

View script ☐

Transfer script after capture ☒

Host

Username

Password

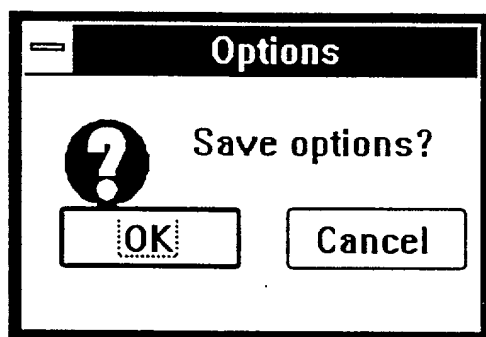
Remote Directory

License Daemon

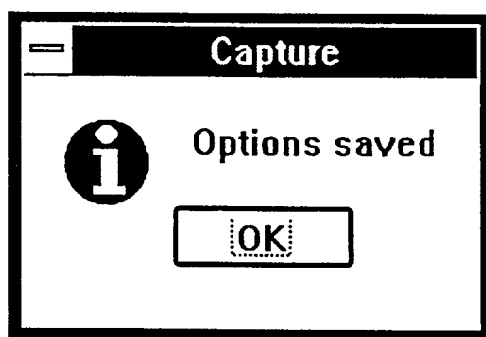
These Options are described more fully in subsequent sections.

When you change an option and select **OK**, a verification window will appear asking you to confirm changes.

Example:

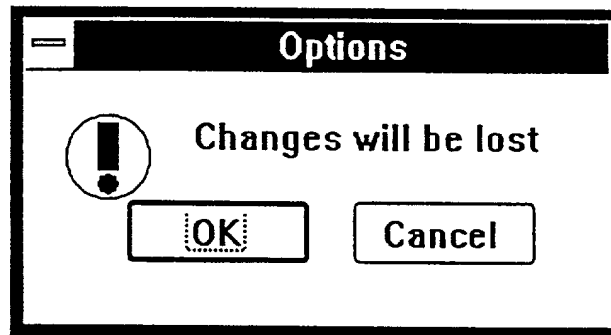


If you select **OK** again, another Window will appear stating that the Options have been saved. Select **OK** to return to the Capture command window.



To ignore any changes you made, select **Cancel**. A verification window will appear stating that the changes will be lost. Select **OK** to return to the Capture command window.

Example:



In the **Options** window, you may change any of the Capture options during your Capture session, except **Database traffic only** and the **Database . . . Chooser**.

4.2.1.1 Think Threshold

This option specifies the number of seconds that EMPOWER/CS will wait before inserting a `Think()` function into the script .c file. (Refer to Section 7 Script Content and Enhancement for a more detailed explanation of think time functions.) For example, suppose you specify a Think threshold of 2 seconds. If no activity is captured after two seconds, EMPOWER/CS will insert a think time function of at least two seconds and the time elapsed until the next action occurs.

You may specify any two digit number of seconds for this option.

4.2.1.2 Bring to Front with

Once you have begun capturing activity, the Capture window will minimize into the lower left corner of the Windows desktop. (*Note:* If you have maximized the window of your test application, you will not be able to see the Capture icon.) By simply pressing a hot key, you can bring the Capture command window up, temporarily halting the session.

With this option, you may specify one of the function keys (F1, F2, etc.) as a hot key. The default hot key is F12.

Example:

Capture Options

Think Threshold

Bring to front with

F12	↓
F10	↑
F11	
F12	↓

View script ☒

Transfer script after ca ☒

Database traffic only ☐

Insert timers ☐

Database...

Host

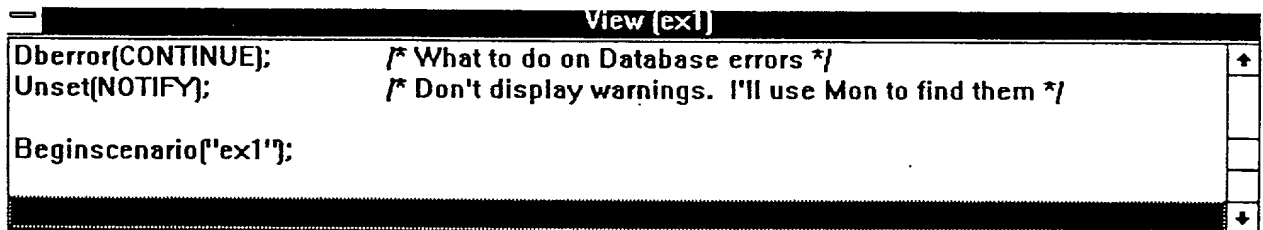
Note: You should specify a function key that will not be used in your application.

4.2.1.3 View Script

Selecting the **View script** option allows you to look at the script file as you Capture. During Capture, the View script window appears on screen showing the script .c

file as user and database activity is captured. This option will help to familiarize you with the script functions that relate to activity you capture.

The following example demonstrates a View script window during Capture:



```
View (ex1)
Dberror(CONTINUE);      /* What to do on Database errors */
Unset(NOTIFY);          /* Don't display warnings. I'll use Mon to find them */

Beginscenario('ex1');
```

4.2.1.4 Database Traffic Only

If you select this option, only database traffic will be captured into the script, which makes your script more compact.

Database traffic includes such functions as `Open()`, `Parse()`, `()`, `Exec()`, `Fetch()`, etc. which are inserted into the script when the client application interacts with the database. No user interactions, such as mouse button presses or `Type()` functions, are inserted into the script file. However, the amount of time taken to type characters or move the mouse will be included in the `Think()` times that are recorded into the script. (Refer to Section 7 Script Content and Enhancement for a full description of these script functions.)

If you specify this option, your script can be executed only in Non-Display mode.

4.2.1.5 Insert Timer

If you specify this option, `Begintimer()` and `Endtimer()` functions are inserted automatically into the script during Capture to mark database traffic. These functions will be inserted around database traffic that occurs between two user events. A user event such as pressing the Return key on the keyboard or

activating a pushbutton on screen may initiate database activity. The parameters to these functions will designate the last user event before database activity began.

When the script is executed, the `BeginTimer()` and `EndTimer()` functions will be time stamped in the script's log file for the Report tool to measure response time of the database activity.

The following example demonstrates `BeginTimer()` and `EndTimer()` captured into a script file around a database query. Notice the user event captured as the functions' parameters was a `ButtonPush()` event, when a user pressed the button "Payable":

```
CurrentWindow("Accounts",147,66,533,360);
ButtonPush("Payable",267,213);

AppWait(0.05);
WindowRcv("SfPtCwCwCwCwCwCwCwCwCwCwCwCw");

BeginTimer("Accounts_Payable");

Dbset(CUR1,DEFER,TRUE);

Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD,
ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);

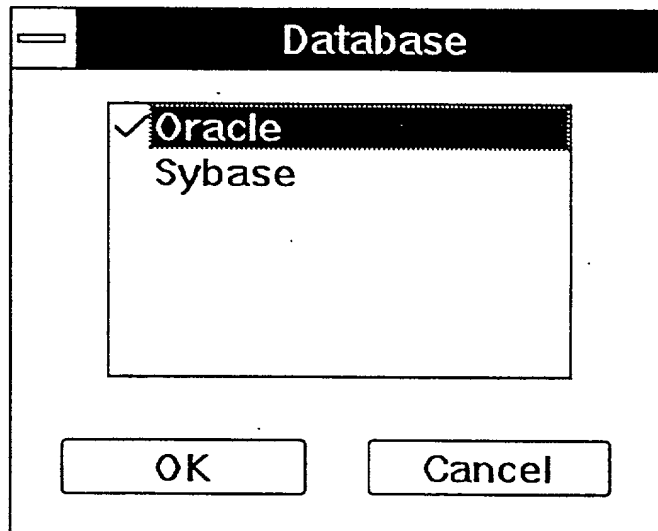
...

EndTimer("Accounts_Payable");
```

4.2.1.6 Database Chooser

This option allows you to choose the database on the server that you will access during the Capture session.

A window similar to the following opens when you select this option:



4.2.1.7 Transfer Script after Capture

Once a script is captured, it must be transferred to the UNIX machine to be executed. If you select this option, your script .c file will be transferred automatically when you stop Capture.

You must specify the following information for the PC to connect to the UNIX script driver and transfer the script file.

Example:

Transfer script after capture ☒

Host

Username

Password

Remote Directory

License Daemon

OK Cancel

In the Options window, enter the **Host** name (the name of the UNIX driver), the **Username**, the **Password** (the user's password, if any), and the **Remote Directory** (the path on the UNIX driver where you wish to transfer the script).

If large amounts of data (i.e., images, large text files, etc.) are input to the database during Capture, such data is inserted into separate data files. These data files will be transferred to the UNIX driver machine with the script if the **Transfer script after capture** option is selected. The data files are associated to the script with a .d extension and a number that is incremented for each file. For example, two data files created for the script, `script1.c`, would be named as follows:

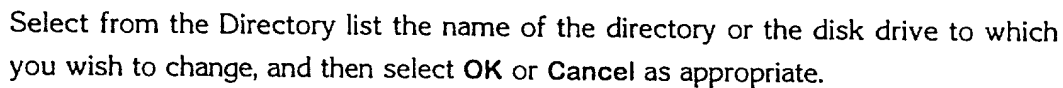
```
script1.d01
script1.d02
```

If you prefer, you may transfer the script file manually with the Transfer tool under EMPOWER/CS Tools. Refer to Section 8 EMPOWER/CS Tools for more information on manual script transfer.

If you specified a license daemon during EMPOWER/CS installation, this option lists the name of that license daemon. If you need to change or specify a license daemon machine (which is the UNIX script driver used for your emulation), you must enter the new name in this dialog box.

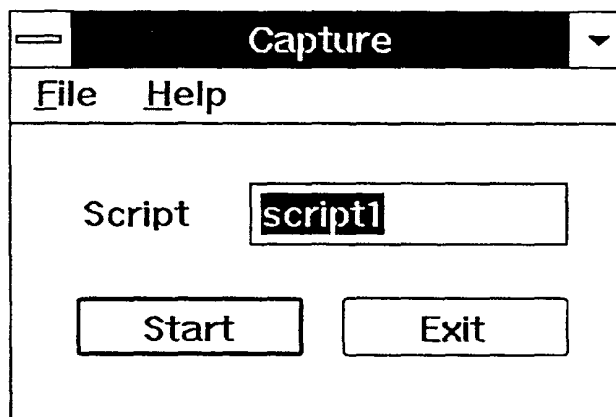
You may use this File Menu option to store scripts in a different directory on your PC.

Select **Directory...** under the **File** Menu. The following window will open designating the directory and disk drive where your scripts currently are being saved.



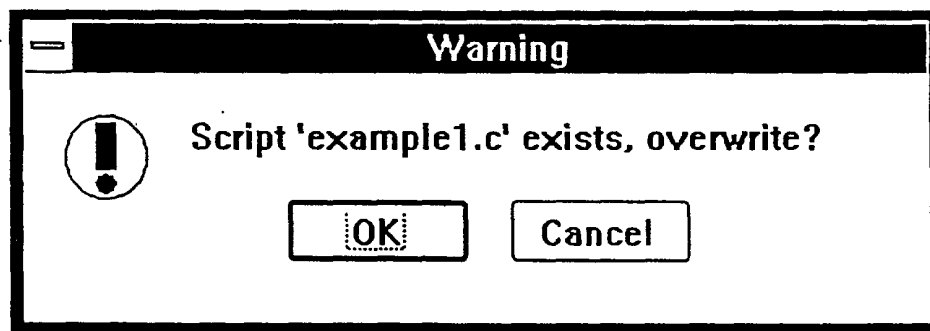
In the Capture command window, enter the script name you are capturing. The script file name can include up to eight alphanumeric and underscore characters. If you enter more than eight characters or other character types, Capture will ignore them. During Capture, the script file is stored automatically in the specified directory in an ASCII file with a ".c" extension.

Example:



When you have entered a script name, select **Start**.

If the script name you enter already exists, the following warning message will appear, asking if you wish to overwrite the existing file. Select **OK** or **Cancel** as appropriate.



Upon selecting **Start**, the Capture window will minimize, or iconify, into the lower left corner of the desktop signifying that all subsequent user and database actions are being captured.

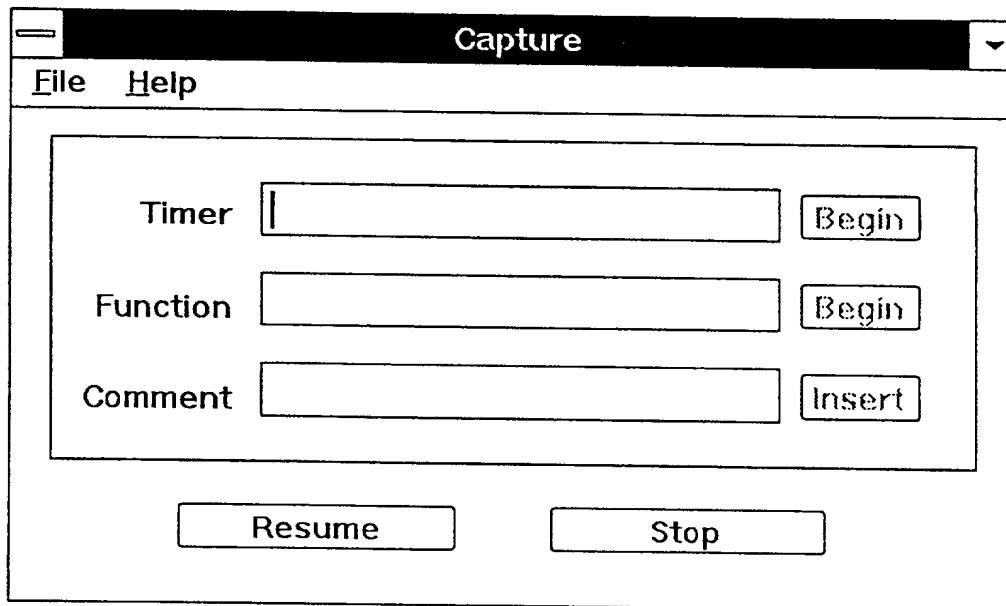
Now, you should open the application to be tested. Refer to your application's user guide for full operating instructions. Remember to maximize your application window upon activation if possible and use keyboard commands instead of the mouse during your Capture session.

Please note that user activities are captured only from Windows standard applications. For example, EMPOWER/CS does not capture user activity from DOS applications running under Windows.

At this time, you should perform the application activity you wish to test.

4.2.4 Comments, Functions, and Timers

You can add timers functions, or comments, to your script file while you are capturing by activating the Capture icon. The following Capture command window will open. When the Capture window appears on screen, all capturing activity halts temporarily.



The screenshot shows a window titled "Capture" with a menu bar containing "File" and "Help". Inside the window, there are three rows of input fields and buttons:

Field Label	Input Field	Action Button
Timer	<input type="text"/>	Begin
Function	<input type="text"/>	Begin
Comment	<input type="text"/>	Insert

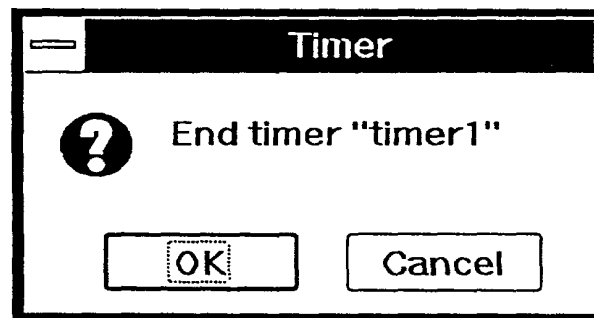
At the bottom of the window, there are two buttons: "Resume" and "Stop".

Inserting timers allows you to mark specific activity for response time measurement. C language functions most commonly are used to help define a script segment for looping purposes. They also are used to define a common interaction, such as logging onto or off of the SUT. You should add comments to your script file to add context to the script for editing purposes.

4.2.4.1 Inserting Timers

You can manually insert the functions `BeginTimer()` and `EndTimer()` into your script to mark activity for response time measurement. You must enter the name of the timer in the **Timer** dialog box of the **Capture** window and select **Begin**. Select the **Resume** button to return to the Capture state to capture the activity you wish to measure.

When you have captured all needed activity, activate the Capture icon to open the **Capture** window. Select the **End** button next to the **Timer** dialog box. The following window will appear to verify the end of the timer:



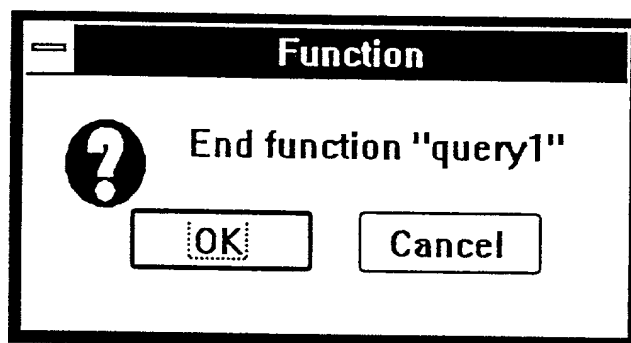
If you have finished inserting the timer, choose **OK** in this verification window. Your desktop then will return to the Capture state.

4.2.4.2 Inserting Functions

To create a C function, enter the name of the function in the **Function** dialog box and select **Begin**. Select the **Resume** button to return to the Capture state to capture the function activity.

After you have completed capturing the function, activate the Capture window and select the **End** pushbutton next to the **Function** dialog box.

The following window will appear verifying the end of the function:



If you have finished inserting the function, choose **OK** in this verification window. Your desktop then will return to the Capture state.

When you create a function, a function call is inserted within the script and the actual function (that includes captured activity) is placed at the end of the script file. Automatically, the EMPOWER/CS functions `Beginfunction()` and `Endfunction()` are inserted around the captured function. After a script is executed, `Beginfunction()` and `Endfunction()` cause time stamps to be recorded in a log file which is used to create detailed response time reports. Marking functions in such a way allows you to measure response time for specific activities in your application. Refer to Section 6.4 The Log File in this manual and to Section 3.3 of the *Multi-User Testing* manual for more information on time stamps.

The following example demonstrates a function `logout()` that was captured into the script file, `script1.c`:


```
logout()
{
  Beginsource("script1.c");
  Beginfunction("logout");

  Think(4.66);

  LeftButtonPress(202,99);

  LeftButtonPress(236,244);

  AppWait(0.28);
  WindowRcv("ScDwAc");

  CurrentWindow("Capture - script1",21,690,57,726);

  Commit(LOG1);

  WindowRcv("Pt");
  Logoff(LOG1);
  Closenv(ORACLE);

  Endfunction("logout");
  Endsource();
}
```

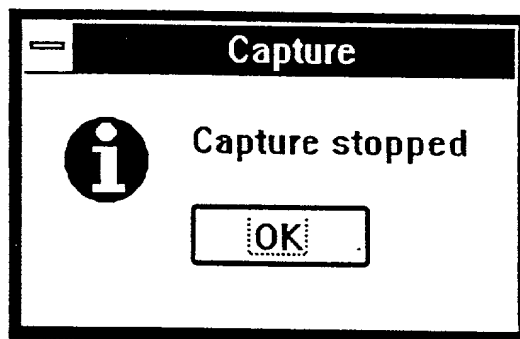
`Beginsource()` and `Endsource()` statements are automatically inserted around `Beginfunction()` and `Endfunction()` to prepare the function as a source file. For instance, during a multi-user emulation, you may wish to break the function out of the script into its own separate file that could be called by multiple scripts performing the same function. Modular script design is achieved by storing one or more functions in separate script source files. The C language `cc` compiler allows functions stored in these files to be compiled separately and then linked to the primary script file. `Beginsource()` and `Endsource()` specify the source file used during script execution.

You may insert a C comment into your script by entering the comment in the **Comment** dialog box of the **Capture** window and selecting **Insert**.

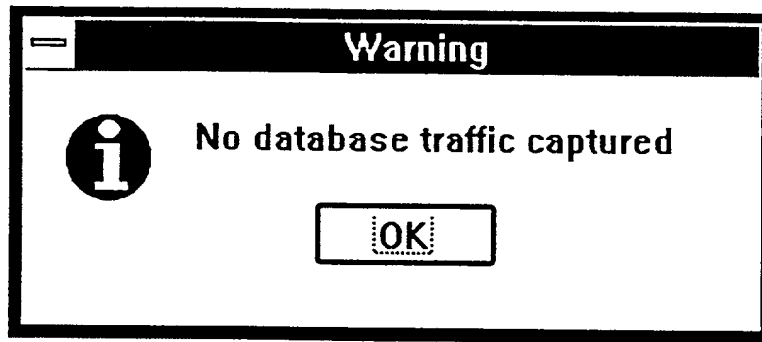
4.2.5 Completing Your Capture Session

Activate the Capture icon to halt the Capture session. The Capture window will appear.

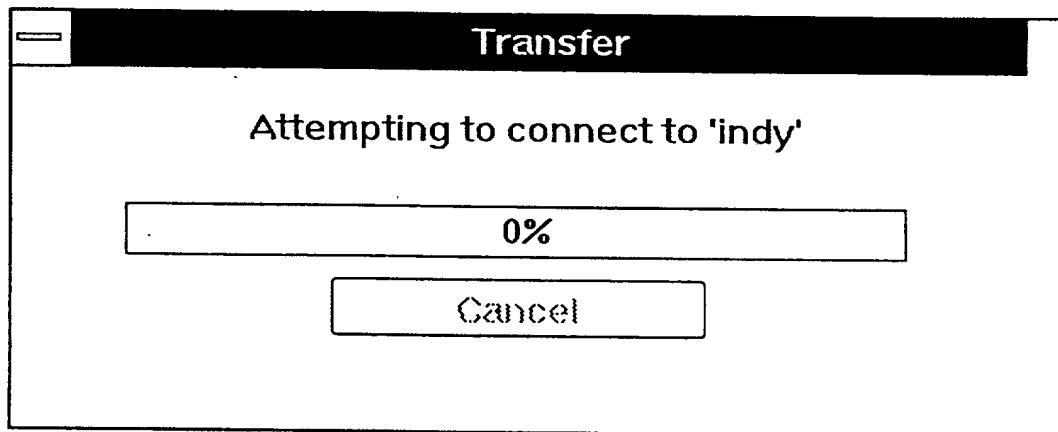
If you are ready to end the Capture session, select the **Stop.** button or **Exit** from the **File** menu. The following window will appear verifying that Capture has stopped:



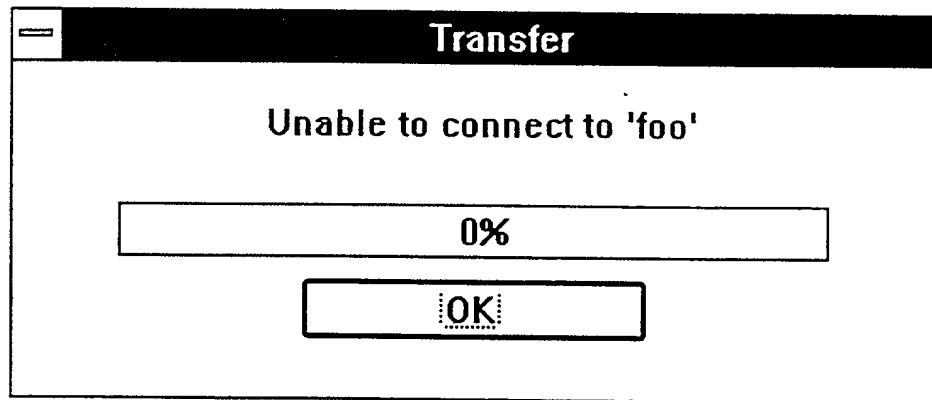
If you did not connect to the SUT to capture database traffic during your Capture session, the following warning message will appear as soon as you stop Capture.



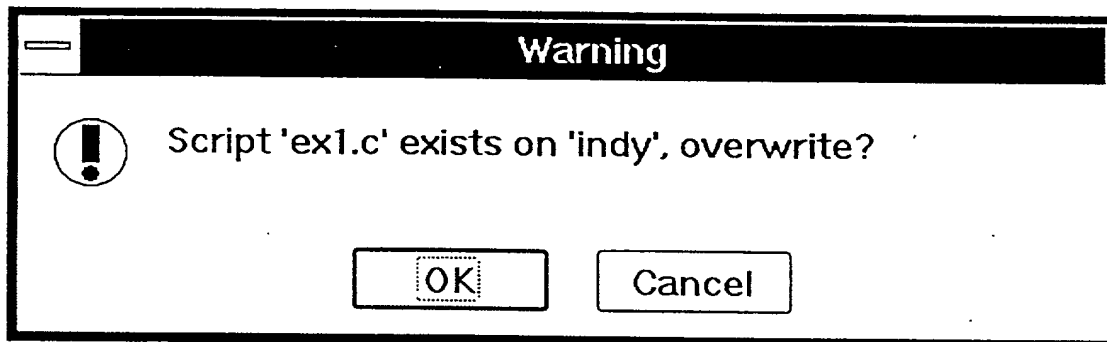
If you chose the automatic transfer option, your script will transfer automatically to the UNIX script driver after **OK** is selected in the **Capture Stopped** window. A window similar to the following appears that shows the progress of the file transfer:



If the PC is unable to connect to the UNIX driver, a message window similar to the following will appear:



If the script .c file already exists on the UNIX script driver upon a successful transfer, the following window will come up asking you to overwrite the existing script .c file. Choose **OK** or **Cancel** as appropriate:



If you did not select the automatic transfer option, you may transfer the script manually using the EMPOWER/CS Transfer Tool (see Section 8.0 EMPOWER/CS Tools) or a third party File Transfer Protocol (FTP) software.

The Capture session is now complete. At this point, you can begin to capture another script or you can close Capture by selecting "Exit" from the File Menu.

2025

[This page intentionally left blank]

```
$ csc -
```

EMPOWER/CS V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

Usage:

```
csc [-EFOacghsvm] [-o ofile] script ...
```

Options:

-E	Preprocesses the script and writes C to stdout
-F	Inserts function declaration (implies -c option)
-O	Optimize script binary
-a afile	Create an archive (.a file) from the named files
-c	Compile but do not link (creates .o file)
-g	Includes symbol table in script binary for debuggers
-h	Excludes help information from the script binary
-s	Prevents strip of script binary
-v	Print verbose cc command line
-m	Excludes Monitor code from script binary
-o ofile	Name output binary

Notes:

Before running CSCC, set the environment variable EMPOWER to the directory that contains the EMPOWER software.

Set the environment variable E_CFLAGS to any additional cc compile options you want.

Cscc uses several environment variables. The EMPOWER/CS variable must be set to the name of the directory containing the EMPOWER/CS software before you can execute Cscc.

Example:

```
$ setenv EMPOWER /usr/empower
```

The EMPOWER/CS directory contains four sub-directories: `bin`, `lib`, `h`, and `install`. Csc will search the `lib` and `h` directories for files needed during script compilation. If you have not set the EMPOWER/CS variable, Csc will search the current directory for the files it needs and produce an error message similar to the following:

can't find empowerm.h

CscC permits you to use a temporary directory other than the current directory for C compilations. Set E_TMPDIR equal to the name of the temporary directory.

Example:

```
$ setenv E_TMPDIR "/tmp"
```

Also before you can execute Csccl, certain environment variables must be set to specify the databases being used. The environment variable E_DATABASES must be set to designate which database with which you are going to compile. Valid choices for Oracle and Sybase databases are ORACLE7, SYBASE4, or SYBASE10. *Note:* If you are not sure which version of Sybase was used during Capture, you can look in your .c file for the Openenv() function. The second parameter will list the appropriate version.

The file `newsript.c` can then be compiled with Csccl:

```
$ csc newscript
Csc: EMPOWER/CS V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95
cc -s -o newscript newscript.c /usr/local/lib/*.a
```

Since EMPOWER/CS scripts are C language programs, defining a common interaction in a C language function often is useful. This allows a script to include a function call rather than the complete set of script entries for activities that you want to be repeated within the script. For example, the following script starts with the `EmpowerCS()` function. The `EmpowerCS()` function is similar to `main()` in a normal C language program. This function calls the function, `function1()`, which is located later in the script.

```
EmpowerCS(argc, argv)
    int argc;
    char **argv;

{
    Thinkuniform(1, 2.5);
    Timeout(300, EXIT);
    Unset(NOTIFY);

    Beginscenario("script1");

    InitialWindow(0, "Desktop", 0, 0, 1024, 768);
    InitialWindow(1, "Program Manager", 830, 186, 73, 605);
    InitialWindow(2, "Accounting", 413, 679, 1029, 771);

    AppWait(0.06);
    WindowRcv("CwPtPt");
}
```

(continued on following page...)


```

LeftButtonPress(457,617);

WindowRcv("DwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt");
CurrentWindow("Run",429,491,880,764);
Type(:c:\\acct\\acct^M");

function1();

...

CurrentWindow("Capture - script1",21,690,57,726,"Min");
Commit(LOG1)

Close(CUR1);

WindowRcv("Pt");
Logoff(LOG1);
Closenv(ORACLE);

Endscenario("script1");
}

function1()
{
    Beginsource("script1");
    Beginfunction("function1");

    AppWait(0.22);
    WindowRcv("CwCwCwCwCwCwCwCw");
    Openenv(ORACLE,VERSIONV6V7);

    Username(LOG1, "scott/tiger@SUT");
    Password(LOG1, "");
    Logon(ORACLE, LOG1);

    Open(LOG1,CUR1);

    WindowRcv("AcSf");
    CurrentWindow("Accounting Application",189,82,685,449);

    Endfunction("function1");
    Endsource();
}

```


The function syntax can be inserted manually or may result from inserting a function during Capture. (See Section 4.2.4.2 Inserting Functions.)

Your script also can access functions that are stored in other script files. Modular script design is achieved by storing one or more functions in separate script source files. The C language `cc` compiler allows functions stored in these files to be compiled separately and then linked.

You must use the `-c` option of the `csc` command to create object files (with an ".o" extension) for each compiled script.

Examples:

```
$ csc -c func1
$ csc -c func2
$ csc -c func3
```

For a script to access these functions, a function call must be inserted within the script and the object files must be linked with the source script. This is accomplished by compiling the script source file and listing the object files as parameters of the `csc` command, as shown below:

```
$ csc script1 func1.o func2.o func3.o
```

In the above examples, three separate functions are stored in the script source files `func1.c`, `func2.c`, and `func3.c`. Each is compiled separately and then linked to the source script file, `script1.c`.

Individual object files may be stored and used by other scripts or included in an EMPOWER/CS library.

When the `-c` option of the `csc` command is used, the function stored in a separate file must include standard C language function formatting (e.g., braces, functions, etc.).

If a script is compiled from multiple source files, each source file must be specified as a source file by including `Beginsource()` and `Endsource()` statements. When editing your script, you should insert the `Beginsource()` function at the entry point of the source file, typically just before the first executable statement in each function. You should insert the `Endsource()` function at the exit point of the file, typically just after the last executable statement in each function.

The following example script segment demonstrates a typical source file:

```
logon1()
{
Beginsource("logon1");

AppWait(0.17);
WindowRcv("SfPtCwCwCwCwCwCw");
Openenv(ORACLE,VERSIONV6V7);

Username(LOG1, "scott/tiger@SUT");
Password(LOG1, "");
Logon(ORACLE, LOG1);

WindowRcv("SfAcSfDwPtPtPtPtPtPt");

CurrentWindow("Accounting",413,679,1029,771);

Open(LOG1,CUR1);

WindowRcv("AcSf");

CurrentWindow("Accounting Application",189,82,685,449);

Endsource();
}
```

Note: If you define C functions during your Capture session, `Beginsource()` and `Endsource()` are placed around `Beginfunction()` and `Endfunction()` statements in the event you wish to break the function out later into a separate source file.

The `-F` option of the `csc` command extends the capabilities of modular script compilation to handle functions that do not include standard C function formatting. This option places the required formatting into the script automatically.

For example, elements of the `logon1()` function may be contained in the following script file called `logon1.c`:

```

Beginsource("logon1");

AppWait(0.17);
WindowRcv("SfPtCwCwCwCwCwCw");
Openenv(ORACLE,VERSIONV6V7);

Username(LOG1, "scott/tiger@SUT");
Password(LOG1, "");
Logon(ORACLE, LOG1);

WindowRcv("SfAcSfDwPtPtPtPtPtPt");

CurrentWindow("Accounting",413,679,1029,771);

Open(LOG1,CUR1);

Endsource();

```

To compile this file into the object file `login1.o`, use the `-F` option:

```
$ csc -F logon1
```

CscC adds the function definition logically to the source file and compiles it into the object file `logon1.o`. The `-F` option may not be used if the function is required to accept arguments.

An executable script file created with the `-g` option will be rather large and will require a large amount of memory to execute. Therefore, using `-g` is not recommended for scripts executed during a sizable multi-user emulation.


```
$ cscc -h example30
```

If you try to display help information for a script that was compiled with the `-h` option, the following error message will result:


```
cc -s -o script1 ./csccl7366.c /usr/local/lib/*.a
```

```
$ csc -v script1
EMPOWER/CS V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

cc -s -L/opt/oracle/lib -cckr -I/usr/local/h -o script1 ./csccl7377.c
/usr/local/lib/empowercsm.a /usr/local/lib/empowercsMON.a /usr/
local/lib/empowerGV.a /usr/local/lib/oralib.a /usr/local/lib/sybstub.a
/usr/local/lib/syb4stub.a -locic /opt/oracle/lib/osntab.o -lsqlnet
-lora -lsqlnet -lcvt6 -lcore -lnlsrtl -lcore -lsocket -lnsl -lm
```

```
csc: can't open example1.c.
```


[This page intentionally left blank]

005204-1552550

.....

Two methods are available for executing a script: Non-Display and Display modes. Non-Display mode executes what was captured between the PC and the SUT but involves only the UNIX script driver and the SUT. Display mode involves the PC, the UNIX driver, and the SUT and displays the captured client/server activity on the PC.

If you did not specify `-h` in the `cscd` command, you can access syntax help for a compiled script by entering the script executable name followed by a hyphen (-).

Example:

```
$ example1 -
example1:  EMPOWER/CS V1.0.0, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95
Usage:
    example1 [-d hostname] [-S scriptid] [log [arg1 arg2 . . .]]
Options:
    -d hostname      Displays script execution
    -S scriptid      Changes scriptid from example30 if not run from MIX
    log              Identifies the log file to be created
    arg1 arg2 . . .  Identifies optional script arguments

Notes:
    If a log is not specified, the log example1.1 will be created.
    If you specify " " as the log, no log will be created.
    You must specify a log if you want to specify arguments.
    arg1 is accessible as the variable argv[3] in the script.
```

Table 1 Demographic characteristics of study population

Executing a script in Non-Display mode requires only the UNIX script driver and the SUT. The script on the UNIX driver replaces the PC to interact with the SUT, and the SUT responds as if it is servicing requests from an actual PC.

To execute a script in Non-Display mode, you must first compile it on the UNIX script driver with the `csc` command.

To execute the script `example30`, type the following at the command line:

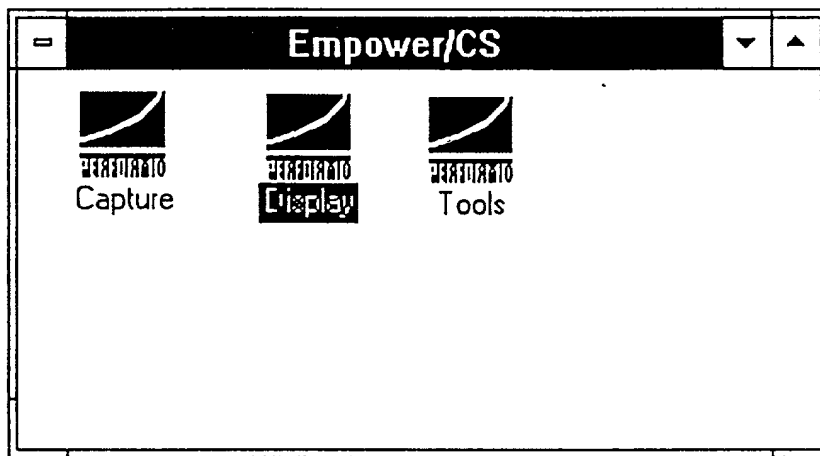
6.2 Script Execution in Display Mode

In Display mode, script execution is initiated from the UNIX script driver. The UNIX driver directs the PC to replay captured user activity and to wait for data returned from the SUT. The PC replays the entire script interacting with the SUT as directed by the UNIX script driver. During script execution, all interaction occurs between the UNIX script driver and PC, and between the PC and the SUT. The SUT receives all PC transmissions and responds accordingly to the PC.

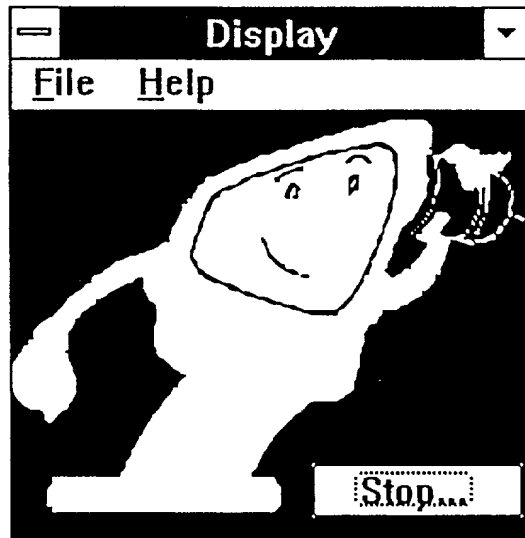
Before a script can be replayed, or executed, in Display mode, you must compile the script on the UNIX driver machine with the `csc` command.

You also must set up the PC to display the script. If your Windows application is not already running, from DOS on the PC, type "win" to start MS Windows.

When the Program Manager appears, select the EMPOWER/CS program group. Then, select the Display program-item icon:



The following window will open that displays a cartoon PC "listening" for network connections from the UNIX script driver:



The UNIX driver must connect to the PC to execute the script and control the captured PC activity. On the UNIX script driver, enter the executable script name with the `-a` option and the name of the PC to which you are displaying.

Example:

```
$ script1 -d vagrant
```

Every user interaction that was captured between the PC and the SUT will execute on the PC. The UNIX driver directs the PC to activate windows, select buttons, enter queries, etc. as if a real user were using the PC. The PC in turn will interact with the SUT as it would if a real user were using it. The SUT will respond to the PC accordingly.

The Display icon on the Windows screen will disappear during script execution and will return only when the script exits.

Example:

```
$ example1 -S example2
```

The `scriptId` variable is used in Monitor and can be used within a script. If you are running a multi-user test that executes a single script several times simultaneously, you should change the Script ID for each instance of the script. The Script ID acts as an emulated user ID.

Note: If you are using the Mix tool, you do not need to specify the `-s` option to execute a multi-user test. Mix inserts the `-s` option automatically by reading the Script ID from the Mix table. (Refer to the *Multi-User Testing* manual for more information on the Mix table.)

6.3.3 Specifying a Log File

When a script executes, a log file is created in the current directory on the UNIX driver. This log file contains all emulated user activity, SQL requests to the SUT, and all data returned to the PC. To calculate response times, this log file also includes time stamps for the time that such functions as `Beginfunction()`, `Endfunction()`, `Beginscenario()`, and `Endscenario()` were executed.

If you do not specify a name for the log file in the script execution command, the log file will be given the same name as the script with a ".1" extension. You do not need to include the ".1" extension if you specify a log file name.

Example:

```
$ example30 -d vagrant log
```

When you specify a log file name, you can include a path name to save the file in a directory other than the current directory.

Example:

```
$ example30 -d vagrant /user/empower/logs/log001
```

You also can define the `E_LOGDIR` environment variable to specify a directory for log files as shown below:

In the Bourne shell:

```
$ E_LOGDIR=/user/empower/logs;export E_LOGDIR
```

In the C shell:

```
$ setenv E_LOGDIR "user/empower/logs"
```

If you do not wish to create a log file, specify the null string (" ") as part of the script execution command which is useful for performing demos when disk space is limited. If you do not create log files, you will not be able to obtain statistical information for your test.

Example:

```
$ example30 -d vagrant ""
```

6.3.4 Specifying Arguments

Arguments may be passed to the script during execution by specifying them in the script execution command. You must specify log file names prior to specifying arguments, even if you wish to use the default log file name.

Example:

```
$ example30 example30.1 john pass001
```

Refer to Section 7.3 of this manual for more information on script arguments.

6.4 The Log File

Executing EMPOWER/CS scripts results in the creation of a log file for each executed script. If a script is executed so that the created log file has the same name as an existing log file, the new log file will overwrite the existing one. If you are running a multi-user test, you must ensure that each emulated user writes to a different log file.

The log file contains copies of user entries, every EMPOWER/CS function entered, SQL requests to the SUT, data returned to the PC, and time stamps used for performance measurement. Entries in the log file that correspond to lines in the script source file include the characters >>> followed by the line number of the corresponding script file entry.

The following is an example EMPOWER/CS log file:


```

>>>      Log: script1.1
>>>      Date: Fri Jan 20 10:03:02 1995
>>>      Command: script1 -d ergy
>>>      Beginsource("script1")
>>>      4 Typerate(5.00)
>>>      5 Pointerrate(150.00)
>>>      6 Thinkuniform(1.000,2.500)
>>>      7 Seed(28310)
>>>      8 Timeout(300,CONTINUE)
>>>      9 Dberror(CONTINUE)
>>>     10 Unset(NOTIFY)
>>>     12 Beginscenario("script1") 10:03:02.89
>>>     14 InitialWindow(0,"Desktop",0,0,1024,768)
>>>     15 InitialWindow(1,"Program Manager",943,162,186,581)
>>>     16 InitialWindow(2,"Accounting",413,679,1029,771)
>>>     18 LeftButtonPress(448, 692)
>>>     21 AppWait(0.05)
>>>     22 WindowRcv("CwPtPt")
CwPtPt
>>>     24 LeftButtonPress(459, 616)
>>>     26 WindowRcv("DwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt")
SfSfAcPtDwCoCwCwCwCwCwCwCwCwCwSz
>>>     28 CurrentWindow("Run",429,491,880,764)
>>>     30 Think 1.923
>>>     33 LeftButtonPress(470, 575)
>>>     35 Type("c:\acct\acct^M")
>>>     37 AppWait(0.22)
>>>     38 WindowRcv("CwCwCwCwCwCwCw")
>>>         Username(LOG1, "scott/tiger@SUT");
>>>         Password(LOG1, "");
>>>         Logon(ORACLE, LOG1);
CwCwCwCwCwCwCw
...

>>>     97 Think 2.141
>>>     98 ButtonPush("Employee Records|Next",739,438)
>>>     99 ButtonPush("Employee Records|Next",739,438)

```

(continued on following page ...)

The UNIX script driver drives the PC, instructing it to activate buttons and keys, to draw windows, to enter data, etc. according to functions in the script. In the above script, those types of user interactions are marked with line numbers which refer back to a particular line in the script .c file. Because this script was executed on the PC in Display mode, you will notice that some lines in the log file contain no numbers corresponding to lines in the source script.

When a script is executed in Display mode, database traffic is generated by the PC, not the UNIX script driver. This traffic is captured and sent to the UNIX driver to be logged in the log file, but because these database functions were not executed by the UNIX driver, they contain no line numbers.

EMPOWER/CS-V1.0.1


```
>>>      Log: script1.1
>>>      Date: Fri Jan 20 10:12:23 1995
>>>      Command: script1
>>>      Beginsource("script1")
>>>      4 Typerate(5.00)
>>>      5 Pointerrate(150.00)
>>>      6 Thinkuniform(1.000,2.500)
>>>      7 Seed(28544)
>>>      8 Timeout(300,CONTINUE)
>>>      9 Dberror(CONTINUE)
>>>     10 Unset(NOTIFY)
>>>     12 Beginscenario("script1") 10:12:23.94
>>>     14 InitialWindow(0,"Desktop",0,0,1024,768)
>>>     15 InitialWindow(1,"Program Manager",943,162,186,581)
>>>     16 InitialWindow(2,"Accounting",413,679,1029,771)
>>>     18 LeftButtonPress(448, 692)
>>>     21 AppWait(0.05)
>>>     22 WindowRcv("CwPtPt")
>>>     24 LeftButtonPress(459, 616)
>>>     26 WindowRcv("DwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt")
>>>     28 CurrentWindow("Run",429,491,880,764,Nor)
>>>     30 Think 1.288
>>>     33 LeftButtonPress(470, 575)
>>>     35 Type("c:\acct\acct^M")
>>>     37 AppWait(0.22)
>>>     38 WindowRcv("CwCwCwCwCwCwCw")
>>>     39 Openenv(ORACLE1, V6|V7)
>>>     41 Username(LOG1, "scott/tiger@SUT")
>>>     42 Password(LOG1, "")
>>>     43 Logon(ORACLE, LOG1)
>>>     ...

>>>     97 Think 2.026
>>>     98 ButtonPush("Employee Records|Next",739,438)
>>>     99 ButtonPush("Employee Records|Next",739,438)
>>>    100 ButtonPush("Employee Records|Next",739,438)
>>>    101 ButtonPush("Employee Records|Close",718,157)
>>>    103 WindowRcv("SfPtDwAcSf")
>>>    105 CurrentWindow("Accounting Application",189,82,685,449)
>>>    107 Think 1.008
```

(continued on following page...)

Now that you have learned the basic concepts for emulating captured activity by executing a script, you are ready to begin a multi-user emulation. However, before proceeding to the *Multi-User Testing Manual*, you may wish to edit or enhance your scripts to emulate more realistic loads on your client/server system. The following section, Script Content and Enhancement, discusses common script functions, methods for editing your scripts, and advanced emulation techniques that allow you to develop realistic scripts.

EMPOWER/CS-V1.0.1

Because your script .c file is a C language program and contains EMPOWER/CS functions, it can be edited to provide additional control over script execution. This section describes EMPOWER/CS script functions, various methods for editing your scripts, and advanced techniques for developing sophisticated scripts.

Because extensive use of C language statements in EMPOWER/CS scripts constitutes advanced load testing, we recommend that you become proficient with all EMPOWER/CS tools before applying the concepts presented in this section.

The following example is a typical EMPOWER/CS script (*Note:* Some content is left out for brevity):

(continued on following page...)

(continued on following page...)


```
Define(CURL, "5", CHAR, 21);
Define(CURL, "6", CHAR, 21);
Define(CURL, "7", CHAR, 16);
Define(CURL, "8", CHAR, 16);
Define(CURL, "9", STRING, 40);
Define(CURL, "10", STRING, 40);
Define(CURL, "11", CHAR, 241);
Exec(CURL);

Dbset(CURL, FETCHSIZE, 64);
Fetch(CURL);
while (GetNextRow(CURL) != NOMOREROWS);

Endtimer("Accounting_Application_Customers");

WindowRcv("AcSfSfPt");

CurrentWindow("Employee Records", 73, 82, 790, 506);

Think(18.13);
ButtonPush("Employee Records|Next", 739, 438);

AppWait(2.57);
WindowRcv("SfPt");

Think(7.80);
ButtonPush("Employee Records|Next", 739, 438);
ButtonPush("Employee Records|Next", 739, 438);
ButtonPush("Employee Records|Next", 739, 438);
ButtonPush("Employee Records|Close", 718, 157);

WindowRcv("SfPtDwAcSf");

CurrentWindow("Accounting_Application", 189, 82, 685, 449);

Think(4.23);

LeftButtonDown(204, 105);
LeftButtonUp(211, 241);

AppWait(0.39);
WindowRcv("ScDwAcSf");

CurrentWindow("Accounting", 413, 679, 1029, 771);

Begintimer("Accounting");
Commit(LOG1);
```

(continued on following page ...)


```
Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE1);

WindowRcv("Pt");

Endtimer("Accounting");

Endscenario("script1");
```

Descriptions of general EMPOWER/CS script functions are presented in the following sections.

7.1.1.1 Begin/End Functions

EMPOWER/CS has three sets of "Begin" and "End" functions which mark the start and finish of scenarios and functions. These functions are `Beginscenario()`, `Endscenario()`, `Beginfunction()`, `Endfunction()`, `Begintimer()`, and `Endtimer()`.

The parameter for each function is the name of the script section being defined. Each "Begin" function must have a corresponding "End" function and the parameter used in each "Begin" function must match the name used in the "End" function.

The `Beginscenario()` and `Endscenario()` functions cause scenario time stamps to be recorded in the log file. These functions are inserted into the script .c file automatically during Capture to mark the beginning of the script. Scenarios typically define large sections of emulated activity. Usually an entire script represents a scenario.

The `BeginTimer()` and `EndTimer()` functions are inserted into the script automatically during Capture if the **Insert Timer** option is specified or manually by the EMPOWER/CS user. They are used to calculate response times for the activities they define. If **Insert Timer** is selected, `BeginTimer()` and `EndTimer()` are placed around database traffic that occurs between two user events in the script. If **Insert Timer** is not selected, the user can insert these functions to specify activity to be measured in the script.

A high fidelity emulation of PC user activity should include a simulation of the time required to type at the keyboard. EMPOWER/CS allows you to specify a typing speed, measured in characters per second, that is applied to all emulated keyboard activity.

The `TypeRate()` function sets the typing speed. During script execution, each time that the emulated user is supposed to be typing at the keyboard, the script will pause a length of time defined as the number of characters to be typed times the type rate specified. For example, if the emulated user must type twelve characters and the type rate is specified as six characters per second (`TypeRate(6)`) then the script will pause two seconds when the emulated characters are to be "typed."

The default function, `Typerate(5)`, is placed at the top of every script created by EMPOWER/CS, specifying a type rate of 5 characters per second. You may change this rate during script editing. If you specify a type rate of zero in the script (`Typerate(0)`), then no delay is used when the script emulates a user typing at the keyboard.

The `Pointerrate()` function specifies the points per second that the mouse pointer moves. During script execution, each time that the emulated user is supposed to move the mouse, the script will pause a length of time defined as the number of points moved times the pointer rate specified. For example, if the emulated user points the mouse to 120 locations and the pointer rate is specified as

During script execution, EMPOWER/CS applies the emulated pointer rate by inserting a delay between each point that the mouse should move. In Display mode, you will notice that the mouse pointer will pause during each pointer movement from one specified location on screen to the next.

Think time functions insert pauses in script execution to emulate when a user pauses to think before continuing to interact with the PC. These pauses are represented by the `Think()`, `Thinkactual()`, `Thinkconstant()`, `Thinktne()`, and `Thinkuniform()` script functions. The `Think()` function specifies when a pause should be inserted during script execution. `Thinkactual()` specifies that actual captured think times should be used. The other three functions define the length of the think delay by specifying a think time distribution.

The `Think()` function, which may appear throughout the script, will cause the script to pause its execution according to the most recently defined think time distribution.

During Capture, you may specify the number of seconds that EMPOWER/CS will wait before inserting a `Think()` function into the script file. The `Think()` function is inserted in the captured script file only after the specified time has elapsed. By default, if no activity is captured after two seconds, EMPOWER/CS will insert a think time function of at least two seconds and the time elapsed until the next

action occurs. The parameter of the `Think()` function specifies the number of seconds elapsed.

During script execution, `Thinkactual()` tells the script to use values in the `Think()` functions that were captured during your Capture session. The following example demonstrates using this function within a script. When you edit your script, insert `Thinkactual()` as shown below:

```
/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */

Type(5);           /* Typing delay in CPS */
Pointerrate(150);  /* Pointerrate in PPS */
Thinkactual();     /* Think delay */
Seed(getpid());    /* Seed random number generator */
Timeout(300, CONTINUE); /* What to do if function takes too long */
Dberror(CONTINUE); /* What to do on Database errors */
Unset(NOTIFY);     /* Don't display warnings. I'll use Mon to find them */

Beginscenario("example1");
```

Think time distribution may be defined with any of these three think time functions: `Thinkconstant()`, `Thinktne()`, and `Thinkuniform()`. Varying think time distribution throughout a script is common and, typically, each application will have its own think time distribution. These functions define a distribution of randomly chosen think time amounts and can be inserted more than once throughout the script file when editing your script. Because the script was captured with think time values, a value must be specified in `Think()` functions for script execution (even if the value is zero), regardless of the think time distribution.

`Thinkconstant()` specifies that the think time is the same every time a `Think()` function is encountered. The think time amount is specified by the parameter `constant`. The syntax is:

```
Thinkconstant(constant)
```


Dberror(cond)

The default functions, `Timeout(300, CONTINUE)` and `Dberror(CONTINUE)`, are placed at the top of every script created by EMPOWER/CS. During script execution, `Timeout(300, CONTINUE)` specifies that the script will wait 300 seconds (five minutes) for the expected response, then continue. `Dberror(CONTINUE)` specifies that the script will continue if it encounters a database error. You can edit these functions and/or insert them throughout the script to suit your testing needs.

- If a script times out and continues prematurely, receipt of the "late" response (meaning the response took longer than the specified 300 seconds) likely will throw script synchronization off, which could cause continual timeouts until the script exits. If your SUT is slow, you may need to increase the timeout value.
- When multiple clients are accessing the SUT during script execution, responses to your script(s) or PC (in Display mode) could be delayed.
- During script execution in Display mode, EMPOWER/CS waits for specific `WindowRcv()` patterns. If the expected `WindowRcv()` patterns are not received in the specified time, a timeout will occur.

- The state of the SUT during script execution may not be identical to its state during Capture (e.g., if the script tries to delete a record in the database that already has been deleted) which could cause unexpected responses to be received by the script or PC.
- Processing a database function may take longer than the specified timeout time.

Timeouts and database errors can be identified by observing script execution in Display Mode or Monitor and by inspecting the log file. The following example shows a portion of a log file containing a timeout. In this example, the script timed out on a `WindowRcv()` function and because the timeout condition was `CONTINUE`, the next script function was executed.

```
>>> 27 AppWait(0.05)
>>> 28 WindowRcv("SfPtDwAcPtPtPtPtPtPtPtPtPtAcSf")
SfPtAcAcSfPtDwPt
>>>      Timeout 16:18:12.25
>>>      Needed:PtPtPtPtPtPtPtPtPt
>>>      Buffer:SfSfAcSf
>>> 30 CurrentWindow("Sample Application",189,82,685,449)
>>> 32 LeftButtonDown(213, 105)
>>> 33 LeftButtonUp(212, 105)
```

7.1.1.5 Set and Unset

To aid in creating sophisticated script files, various EMPOWER/CS options are available during script execution. The `Set()` and `Unset()` functions allow you to specify certain activities and settings for your emulation. These functions can be inserted or changed when you edit your script.

`Set()` turns on EMPOWER/CS options during script execution. `Unset()` turns off EMPOWER/CS options during script execution.

Valid options are listed below:

<u>Option</u>	<u>Description</u>
LCMD	log miscellaneous commands
FLUSH	flush SUT responses to the log that are detected after a pattern match in <code>s WindowRcv()</code>
NOBUF	don't use buffered writes to the log file
NOTIFY	display a message when a timeout occurs, execution is suspended, or execution resumes
BELL	ring the bell twice when a timeout occurs
LOGGING	Enable logging

7.1.1.6 Mouse Activity

The following list shows all the functions used to emulate mouse button activity for the left, middle, or right mouse buttons during script execution:

LeftButtonDown(x,y)	MiddleButtonDown(x,y)	RightButtonDown(x,y)
LeftButtonUp(x,y)	MiddleButtonUp(x,y)	RightButtonUp(x,y)
LeftButtonPress(x,y)	MiddleButtonPress(x,y)	RightButtonPress(x,y)
LeftDblPress(x,y)	MiddleDblPress(x,y)	RightDblPress(x,y)

These functions are inserted into the script .c file during Capture when the specified mouse activity is performed.

The function parameters `x,y` indicate the xy coordinates of the mouse on screen at the time the mouse button was activated during Capture.

The "ButtonDown" functions indicate when the specified button on the PC mouse was pressed down.

The following example script segment contains various mouse button activities:

(continued on following page...)


```
AppWait(0.38);
WindowRcv("Pt");

LeftButtonDown(340,216);
LeftButtonPress(333,219)

AppWait(0.17);
WindowRcv("Pt");

LeftButtonPress(350,202);
```

7.1.1.7 Keyboard Activity

The following list shows the functions used to emulate keyboard activity during script execution:

KeyPress (key)	SysKeyPress (key)
KeyDown (key)	SysKeyDown (key)
KeyUp (key)	SysKeyUp (key)

The parameter `key` represents a Microsoft Windows virtual key code value. All possible virtual key code values are listed below:

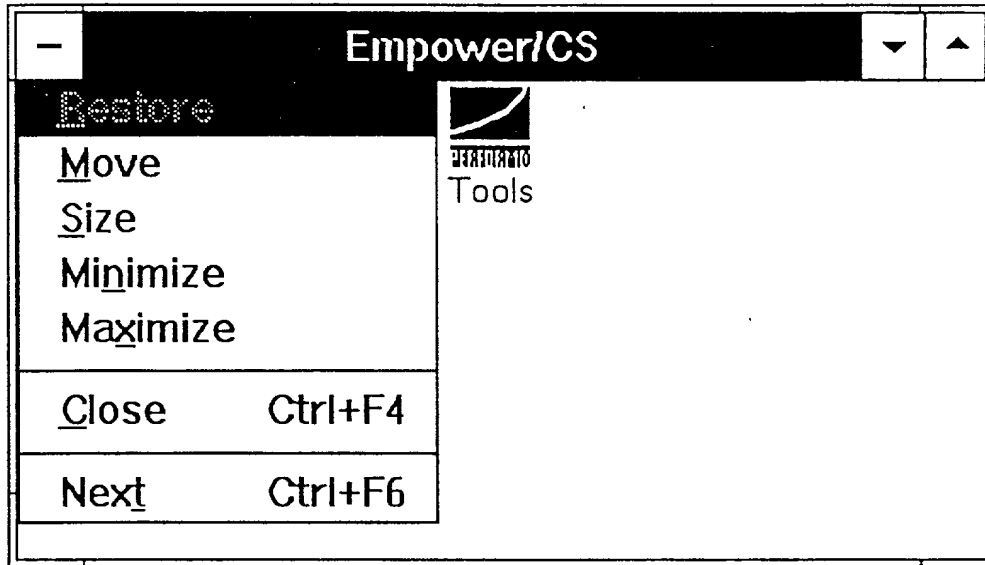
VK_TILDE	VK_LAPPOST	VK_UNDERSCORE
VK_HYPHEN	VK_PLUS	VK_EQUAL
VK_LCURLY	VK_LSQUARE	VK_RCURLY
VK_RSQUARE	VK_COLON	VK_SEMI
VK_DQUOTES	VK_SQUOTES	VK_LTHAN
VK_COMMA	VK_GTHAN	VK_PERIOD
VK_QUESTION	VK_SLASH	VK_PIPE
VK_BKSLASH	VK_CANCEL	VK_CLEAR
VK_TAB	VK_BACK	VK_SHIFT
VK_CONTROL	VK_MENU	VK_PAUSE
VK_CAPITAL	VK_ESCAPE	VK_SPACE

These functions are inserted into the script during Capture when the specified key activity is performed.

A "KeyPress" function is translated into a KeyDown/Up sequence. A KeyPress function is captured into the script when no user events are contained within a KeyDown/Up pair.

The "SysKey" functions indicate that the keyboard activity included pressing the Alt key. "Sys" implies that the keystrokes are being sent to the System Menu of

the current window which is located under the icon in the top left corner of the window as shown below:



During script execution in Display mode, these functions are used to emulate the specified keyboard activity. In Non-Display mode, type rate is applied to these events and the script delays accordingly.

Key events may be edited in your script file, but because you change the expected activity from when it was captured, you may break the script when it is executed.

The following example demonstrates various keyboard and mouse activity in a script file:

```
Beginscenario("example2");  
  
LeftButtonPress(188,381);  
  
WindowRcv("CwPtPtDwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt");
```

(continued on following page...)


```
LeftButtonDown(193,316);
LeftButtonUp(193,317);

CurrentWindow("Run", 189,82,685,449);

AppWait(0.11);
Type("c:\\acct\\c\\acct^M");

WindowRcv("CwAcSfDwPt");

CurrentWindow("General Ledger", 234,99,704,520);

AppWait(1.26);

SysKeyDown(VK_MENU);
SysKeyPress(VK_SPACE);
SysKeyUp(VK_MENU);

Type("scott^Itiger^IFOO^M")
```

7.1.1.8 Type

The `Type()` function is recorded into a script file during Capture when keystrokes are entered from the keyboard. The parameter of the `Type()` function may include the standard keyboard keys pressed (except for those defined as virtual key codes in Section 7.11.2) and the non-displayed key combinations defined below. Each key translates into a `KeyDown/Up` pair for the emulation.

Non-displayed key combinations are shown in the parameter of the `Type()` function using the standard UNIX technique in which the character `^` represents the Control key. Common control sequences captured into the `Type()` function are listed below:

- ^M Carriage Return
- ^H Backspace
- ^I Tab
- ^? Delete

be input to the database, you will need to change the corresponding `Parse()` or `Data()` statements.

7.1.1.9 ButtonPush

The `ButtonPush()` function is inserted into the script file during Capture to indicate that a specific button, a MS Windows pushbutton such as **OK**, **Cancel**, **Yes**, **No**, etc., was activated. It is used during script execution in Display mode to move the mouse to activate the specified pushbuttons. In Non-Display mode, this function simulates mouse movement as defined in the `x,y` parameters to allow for mouse pointer delay.

The syntax of this function is:

```
ButtonPush(str, x,y);
```

The parameter `str` may be listed in a format similar to the following examples:

```
ButtonPush("Program Manager|Run|OK",226,99);
```

```
ButtonPush("Tools|#c1|#c4",68,69);
```

```
ButtonPush("OK",187,201);
```

```
ButtonPush("#c2",299,134);
```

The format of the `str` parameter is designed so that EMPOWER/CS can easily locate the captured pushbutton during script execution in Display mode. This format is based on the MS Windows concept of a tree structure.

The MS Windows tree structure is based on a heirarchy of windows where each window that is accessed from a primary, or parent, window is a child of that parent. The `str` parameter is pipe-delimited, and is listed right to left from child to parent window where the right-most item is the name of the pushbutton. If one of the windows or the pushbutton has no title, something like "#c1" will be listed to

In the first example listed above, OK is the name of the pushbutton that was activated. Run is the name of the window that contained this OK button, and Run was a child window of the parent window 'Program Manager'. In the second example listed above, #c4 was the pushbutton activated and was the fourth child of the first child (#c1) of the Tools window.

If you wish to edit this function in your script file and need to determine a button's tree structure, you can use the Tree Tool under EMPOWER/CS Tools. See Section 8.4 of this manual for more information on the Tree tool and the MS Windows tree structure.

In the following example script segment, the user activated the OK pushbutton to open an application:

```
CurrentWindow("Run",189,82,685,449);
Type("c:\\acct\\c\\acct^M);
ButtonPush("OK",354,153);
```

This EMPOWER/CS function is inserted into the script during Capture after the `Beginscenario()` function. `InitialWindow()` records the state of your Windows desktop by listing all active program windows at the time the Capture session

The following example demonstrates `InitialWindow()` functions captured in a script file:

```
Beginscenario("script1");  
  
InitialWindow(0,"Desktop",0,0,1024,768);  
InitialWindow(1,"Program Manager",989,34,232,453);  
InitialWindow(2,"MS-DOS Prompt",0,0,MAXWIDTH,MAXHEIGHT);
```

The second parameter identifies the title of the window.

During script execution in Display mode, `InitialWindow()` attempts to locate the windows listed as parameters and place them in their captured positions. The function does not apply to Non-Display mode script execution.

If the specified programs are not open at the time of script execution or if a window could not be moved to the specified position, the log file will record a warning but script execution will continue. Therefore, before you execute your script in Display mode, your desktop should have the same applications open as when you captured the script.

Note: The first `InitialWindow()` function listed in the script indicates the resolution of the screen as the first xy parameters are 0,0. You will notice a task order of 0 specified in this first `InitialWindow()` function. The zero indicates to EMPOWER/CS to check the screen resolution making sure the script is displayed in the same resolution as captured. For instance, if a script was captured in vga mode and you attempted to display it in superVGA mode, the captured xy coordinates would be invalid. Therefore, when displaying your script, you should use a screen that is the same resolution as the screen used during Capture.

The `AppWait()` function is inserted into your script file during Capture before a `WindowRcv()` function to record the amount of time taken to draw a window on screen. This function is used during script execution to emulate the application delay for drawing a window on screen. During script execution, this function only applies to Non-Display mode, because in Display mode, windows would actually be activated.

The parameter of `AppWait()` is the time in seconds of the captured application delay. If you wish to change your `AppWait` delay, you may set a multiplication factor with the `AppWaitFactor()` function. This function multiplies the `AppWait` delay times the factor. For instance, during script execution your application may be twice as slow because it is receiving twice as much data from the SUT. You can set the `AppWaitFactor()` to .2 so that the script will emulate the extra delay.

Example:

```
AppWaitFactor(0.20);
AppWait(1.16);
WindowRcv("SfDwAcSfSfSfPt");
```

7.1.1.12 WindowRcv

The `WindowRcv()` function is inserted into the script during Capture when an activated window is drawn on screen and usually follows an `AppWait()` call. Consecutive `WindowRcv()` functions can be listed within the script file.

This function is used during script execution in Display mode to ensure that the activated window is drawn on screen. In Non-Display mode, this function is ignored because the `AppWait()` function emulates the amount of time required to draw the specified window.

The parameters of this function are MS Windows standard two-letter pneumonics that represent the following commands:

<u>Command</u>	<u>Description</u>
Ac	Activate Window
Co	Command
Cw	Create Window
Dw	Destroy Window
Pt	Paint
Sf	Set focus
Sz	System Command


```
AppWait(0.05);  
WindowRcv("CwPtPtDwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt");  
WindowRcv("PtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");  
  
CurrentWindow("Run",429,491,880,764);
```

The `CurrentWindow()` function is inserted into your script file to record all titled windows that were activated during the Capture session. This function is used to ensure windows are in their captured state during script execution in Display mode. It will be listed in Monitor during script execution in Display and Non-Display modes adding context to script execution.

The parameters of `CurrentWindow()` identify the title of the current window (window with focus), define the top left xy coordinates, and then, define the width and height of the window's position on screen. If the width and height are both zero, the window is minimized. If the x,y coordinates are both zero and width is `MAXWIDTH` and height is `MAXHEIGHT`, the window is maximized. If neither of these conditions apply, the window is in a normal state.

The following example demonstrates how `CurrentWindow()` would appear in a script file. The activated window was "Accounting" which appeared on screen in a normal state:


```
AppWait(0.39);  
WindowRcv("ScDwAcSf");  
  
CurrentWindow("Accounting",413,679,1029,771);
```

The following example is a portion of a log file that recorded a warning when a `CurrentWindow()` function could not execute properly because the window did not exist during script execution in Display mode:

```
>>> CurrentWindow("File Manager",0,0,MAXWIDTH,MAXHEIGHT)
Warning: Unable to find window
```

This function should not be edited or removed from your script file because you change the state of your Windows desktop from when it was captured and, therefore, run the risk of breaking the script.

7.1.2 Interacting with the SUT

For you to fully understand the process of interacting with the SUT and the EMPOWER/CS functions associated with this process, the general behavior of the database environment should first be explained. The following paragraphs and sections will introduce you to levels of the database environment that represent specific database communication structures, define the actual script statements that represent such operations as querying or inserting data, and describe other EMPOWER/CS functions that perform various database operations.

7.1.2.1 Communication Structures

In the database environment, certain communication structures are defined to access the database on the server for retrieving, inserting, or changing data. These

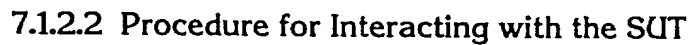
communication structures must be accessed in a particular order as explained below:

- The first level of the database environment is the database environment as a whole. An entire environment is defined that identifies the database being accessed.
- The next level is a log on communication structure. To connect to the database, at least one log on communication structure is defined. If the client application requires multiple, simultaneous connections, one log on structure is defined for each connection.
- The third level of the database environment is the cursor structure. To process SQL requests to the SUT, a cursor communication structure is defined for each SQL request.

In your captured script .c file, you will notice such parameters to various database functions as ORACLE, LOG1, LOG2, CUR1, CUR2, etc. These parameters are identifiers of each level of the database environment that was opened to access the SUT. Each level is numbered to link all database operations particular to a certain environment, log on, or cursor structure. For example, the call `Exec(CUR1)` would execute the `Parse()`, or SQL, statement that belongs to the cursor, CUR1.

The following diagram demonstrates the database environment structure and how the SUT is accessed.

Database Environment



To access and interact with a database on the server during script execution, EMPOWER/CS database functions are called in a particular order. The EMPOWER/CS database functions will occur in the script generally based on the following standard procedure:

Basic Procedure and Script Structure for Database Activity

- ❑ A database environment is opened, represented by the call `Openenv()` in the script which specifies the database to be accessed
- ❑ A connection to one or more databases occurs with a `Logon()` to each specified database
- ❑ One or more cursors are opened, represented by the `Open()` function in the script, to process SQL statements
- ❑ The SQL statements required to perform database operations such as querying, inserting, or deleting data are processed with `Parse()`
- ❑ The select-list items of the SQL statement are described as necessary with `Describe()` functions
- ❑ `Bind()` functions are called for input statements to bind the address of an input variable to each placeholder in the SQL statement
- ❑ For queries, `Define()` is called to define an output variable for each select-list item in the SQL statement
- ❑ `Exec()` is called to execute the `Parse()` and, therefore, the SQL statements
- ❑ For queries, `Fetch()` and `GetNextRow()` are called to retrieve and sort through specified rows of data
- ❑ The cursors are closed with `Close()`
- ❑ A `Logoff()` function is called for each log on structure to disconnect from the specified database(s)
- ❑ Each database environment is closed with `Closenv()`


```
Type("c:\\accts\\accts~M");

AppWait(0.22);
WindowRcv("CwCwCwCwCwCw");
Openenv(ORACLE,VERSIONV6V7);

Username(LOG1, "scott/tiger@FOO");
Password(LOG1, "tiger");
Logon(ORACLE, LOG1);

WindowRcv("SfAcSfDw");

Open(LOG1,CUR1);

WindowRcv("AcSf");

CurrentWindow("Accounts Application",189,82,685,449);

Think(4.77);
ButtonPush("Accounts Application|#c5",355,270);

AppWait(5.38);
WindowRcv("SfCwCwCwCwCwCwCwCwCwCwCwCwCwCwCwCw");
Dbset(CUR1,DEFER,TRUE);

Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1, ADDRESS_LINE_2,
ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS
FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);

Dbset(CUR1,MAXARRSIZE, 64);
Define(CUR1, "1", STRING, 40);
Define(CUR1, "2", CHAR, 21);
Define(CUR1, "3", CHAR, 21);
Define(CUR1, "4", CHAR, 21);
Define(CUR1, "5", CHAR, 21);
Define(CUR1, "6", CHAR, 21);
Define(CUR1, "7", CHAR, 16);
Define(CUR1, "8", CHAR, 16);
Define(CUR1, "9", STRING, 40);
```

(continued on following page...)


```
Define(CUR1, "10", STRING, 40);
Define(CUR1, "11", CHAR, 241);
Exec(CUR1);

Dbset(CUR1, FETCHSIZE, 64);
Fetch(CUR1);
while (GetNextRow(CUR1) != NOMOREROWS);

Commit(LOG1);

Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE);

Endscenario("script1");
```

The following sections further explain the procedure for interacting with the SUT.

7.1.2.3 Open an Environment

As stated in the previous section, the first level of the database environment is the database environment as a whole. This level is represented in a script file by such database names as ORACLE, SYBASE2, etc. and is accessed with the function `Openenv()`. The `Openenv()` function is inserted into the script when a database environment is opened. When executing a script, `Openenv()` opens an environment to specify the database and database version to be used for the emulation.

In some cases, multiple databases may be accessed and activity for each database may occur concurrently, therefore, more than one `Openenv()` call may be listed within a script.

7.1.2.4 Connect to the SUT

The second level of the database environment is the log on level which opens a connection to the SUT through a log on communication structure. This level is defined with a `Logon()` call in the script and is represented in function parameters as a pointer to a log on structure called `LOG1`, `LOG2`, etc. The first log on structure defined will be numbered by EMPOWER/CS as `LOG1`, the second as `LOG2`, etc.

The `Logon()` function is inserted into the script when a log on connection is opened to the database. The parameters to `Logon()` specify to which database the script is connecting and the number of the log on structure.

The script establishes communication with one or more databases by calling a `Logon()` function for each connection. Logging on to the database is necessary for the subsequent database operations of querying and inserting data. Multiple log on structures are possible for each database environment.

To successfully access or log on to the database, a Username and Password also are defined. Therefore, before a `Logon()` function executes, the `Username()` and `Password()` functions are called in the script. `Username()` sets the name of the user who is logging on to the database and the `Password()` function sets the user password to log on to the database. These functions are inserted into the script file when a user name and password are entered to log on to the database.

The following script segment demonstrates the process of logging on to the database, `ORACLE1`, from the log on structure `LOG1`:

```
Type("c:\\accts\\accts^M");

AppWait(0.22);
WindowRcv("CwCwCwCwCwCw");
Openenv(ORACLE1,VERSIONV6V7);

Username(LOG1, "scott/tiger@FOO");
Password(LOG1, "tiger");
Logon(ORACLE1, LOG1);
```


The third level of the database environment is the cursor communication structure. The cursor structure is where the script actually interacts with the database by processing SQL requests to the SUT. Cursors are opened with an `Open()` call in the script file and are represented in function parameters as a pointer to a cursor structure called `CUR1`, `CUR2`, etc. The first cursor opened in a particular log on structure will be numbered by EMPOWER/CS as `CUR1`, the second as `CUR2`, etc. The `Open()` function is inserted into the script when a cursor structure is opened.

The `Open()` function will be listed after the `Logon()` function and before the `Parse()` in your script file.

Parsing the SQL statement associates it with the appropriate cursor in the script. Parsing a SQL statement includes sending it to the SUT, verifying its syntax is

The first parameter of the `Parse()` function identifies the cursor structure that was opened with the associated `Open()` function. The second parameter lists the SQL statement to be parsed.

Because the SQL statement can be executed only from a cursor structure, `Parse()` occurs after an `Open()` function; and, because the SQL statement is executed by the `Exec()` function, `Parse()` occurs before `Exec()` in the script.

```
Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER,
COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");
```

```
Parse(CUR1, "INSERT INTO CUSTOMERS ( ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1, ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS ) VALUES ( 789, 'Robert', 'Adams', '1 Maple Lane', 'Anytown', 'VA, 22222', '555-9856', '555-7634', 500, 700, 'Good Job' )");
```



```
Parse(CURL1, "DELETE FROM CUSTOMERS WHERE ID = 789 AND FIRST_NAME =
'Robert' AND LAST_NAME = 'Adams' AND ADDRESS_LINE_1 = '1 Maple
Lane' AND ADDRESS_LINE_2 = 'Anytown' AND ADDRESS_LINE_3 = 'VA,
22222' AND PHONE_NUMBER = '555-9856' AND FAX_NUMBER = '555-7634'
AND COMM_PAID_YTD = 500 AND ACCOUNT_BALANCE = 700 AND COMMENTS =
'Good Job' ");
```

7.1.2.7 Describe Select-List Items

During script execution, `Describe()` sends a request to the database for a description of specified select-list items in the SQL statement. It returns information about the select-list items needed to determine how to convert, display, or store the data that will be returned when rows are fetched for a query. Such returned information can include the name of the variable, the data type, the size of the item, whether the data is null-terminated or updateable, etc.

EMPOWER/CS-V1.0.1

Example:

```
DescribeAll(CUR1, 1, 12);
```

In this example, all variables listed in the SQL statement from CUR1, starting from position 1 to position 12, will be described.

In some cases, you may notice a `DescribeProc()` function instead of `Describe()`. The `DescribeProc()` function describes all variables used in a specified stored procedure. A stored procedure is an operation that is stored on the database to be executed later. The procedure must be described before a parse is completed. Therefore, this function will occur in the script before `Open()` and `Parse()`.

7.1.2.8 Bind the Addresses of Input Variables

Some SQL statements require that data be input to the database. Placeholders for input variables in the SQL statement mark where specific data must be input and are indicated by leading colons (Example: `:NAME`). If a SQL statement requires data to be input to the database, a `Bind()` function will be inserted into the script to bind each placeholder to a variable that is to be passed to the database. Therefore, when the SQL statement is executed, the database will receive the data that was placed in the input variables.

The parameters of the `Bind()` function identify the variable by the cursor number of the associated SQL statement, the variable placeholder listed in the SQL statement, the variable's data type, and the variable's length in bytes.

The `Bind()` function is listed in the script after a `Parse()` statement and before `Exec()`.

The following example demonstrates a `Bind()` function inserted into a script for the variable, `x`:

```
Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,
DEPTNO FROM EMP WHERE EMPNO=:X");

...

Bind(CUR1, "X", STRING, 10);
```

Note: If you notice a `Bindp()` function in your script, a `Bind()` operation was performed for a variable based on its position in the SQL statement instead of placeholder name. The parameter would specify a position instead of a name. `Bindp()` binds the variable's position to the variable.

If a variable was inserted and then retrieved, the `BindDefine()` function is inserted into the script. `BindDefine()` inputs a variable to the database and then gets a new value. Usually, the value returned is the old value. For example, if the SQL statement specified that the value `Smith` be inserted into a record to overwrite the existing value `Jones`, `BindDefine()` would specify `Smith` as the new value for the record and then return the old value `Jones` to the variable.

7.1.2.9 Define

The `Define()` function is inserted into the script when output variables are defined for storing data to be fetched from the database. The SUT places data in these output variables when a `Fetch()` function is called. `Define()` functions are used only when fetching records from the database for a query.

`Define()` associates the address of an output variable in the program with each select-list item in a SQL query statement. The `Define()` function defines each select-list item in the SQL statement by a cursor number, the item's position in the SQL statement, the variable's data type, and the variable's length. The positions in the SQL statement are defined beginning with 1 for the first (or left-most) select-list

item, 2 for the second, and so on. The `Define()` function is listed in the script after a `Parse()` statement and before `Fetch()`.

The following script segment demonstrates the `Define()` statements defining variables for each select-list item of the SQL statement. Notice in the first `Define()` statement below, the variable position "1" refers to ID in the SQL statement:

```
Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER,
COMM_PAID_YTD, ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);

Dbset(CUR1, MAXARRSIZE, 64);
Define(CUR1, "1", STRING, 40);
Define(CUR1, "2", CHAR, 21);
Define(CUR1, "3", CHAR, 21);
Define(CUR1, "4", CHAR, 21);
Define(CUR1, "5", CHAR, 21);
Define(CUR1, "6", CHAR, 21);
Define(CUR1, "7", CHAR, 16);
Define(CUR1, "8", CHAR, 16);
Define(CUR1, "9", STRING, 40);
Define(CUR1, "10", STRING, 40);
Define(CUR1, "11", CHAR, 241);
Exec(CUR1);
```

7.1.2.10 Execute the Parse

The `Exec()` function executes the operations specified in the associated SQL statement. This function is inserted into your script when a SQL statement is executed on the SUT.

The parameter of `Exec()` identifies the cursor of the associated SQL statement.

After the specified output variables are defined and the SQL statement has been executed, the rows of data that satisfy a query are fetched. In your script, the `Fetch()` function is inserted when the data specified in the SQL statement to satisfy a query is retrieved. During script execution, when the number of rows of data specified in the option `FETCHSIZE` is fetched from the database, the data is placed into the defined output variables in a buffer on your UNIX script driver machine. The parameter of `Fetch()` specifies the cursor structure for the associated SQL statement.

Dbset (curnum, FETCHSIZE, n)

Each `Fetch()` statement returns the next row from the set of rows that satisfies a query. After the last row has been returned, the next fetch will return an error that no remaining rows could be fetched.

EMPOWER/CS-V1.0.1


```
Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,  
DEPTNO FROM EMP, FOR UPDATE OF EMPNO, ENAME, JOB, MGR, HIREDATE,  
SAL, COMM, DEPTNO");
```

```
Exec (CUR1) ;
```

```
Fetch(CUR1);
while (GetNextRow(CUR1)!=NOMOREROWS);
```



```
Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE);

Endscenario("script1");
```

The following sections describe other EMPOWER/CS database functions that may appear in your script file.

Client applications may specify certain options that control the behavior of the database environment. The `dbset()` function is inserted into the script when such options are set according to the client application and the SUT and how they interact. This function may occur throughout a script and it specifies various options for the database environment.

Dbset(num, opt, value)

EMPOWER/CS-V1.0.1

Option	Description
FETCHSIZE	Specifies the number of records to fetch from the database when a <code>Fetch()</code> function is executed. The <code>Dbset()</code> function that sets the <code>FETCHSIZE</code> will occur before a <code>Fetch()</code> . The <code>FETCHSIZE</code> value can not be larger than the value specified for <code>MAXARRSIZE</code> . If the <code>FETCHSIZE</code> was specified as 50 and <code>MAXARRSIZE</code> was specified as 20, <code>EMPOWER/CS</code> will reduce the <code>FETCHSIZE</code> to 20. The default value for this option is 1 which means that if <code>FETCHSIZE</code> is specified as zero or lower, <code>EMPOWER/CS</code> automatically sets the value to 1.
MAXARRSIZE	Sets the maximum array size for retrieving or inserting records. If the array size is set at 20, 20 rows of data can be inserted or fetched. If this option is set at 20, then 20 placeholders are allocated for inserting or fetching 20 rows of data. The <code>Dbset()</code> function that sets the <code>MAXARRSIZE</code> will occur before the <code>Bind()</code> and <code>Define()</code> functions in a script. The default value for this option is 1 which means that if <code>MAXARRSIZE</code> is specified as zero or lower, <code>EMPOWER/CS</code> automatically sets the value to 1.
INSERTSIZE	Specifies the number of rows of data to be inserted into the database. This option will be listed before the <code>Data()</code> function in a script in the format <code>Dbset(CUR1, INSERTSIZE, n)</code> . This option allows array binding. For example, if <code>n</code> is 0 or 1, one row can be inserted at a time into the database. If <code>n</code> is 50, 50 rows will be inserted into the database and 50 <code>Data()</code> lines must be listed for every row before <code>Exec()</code> . The <code>INSERTSIZE</code> value can not be larger than the specified <code>MAXARRSIZE</code> . The default value for this option is 1 which means that if <code>INSERTSIZE</code> is specified as zero or lower, <code>EMPOWER/CS</code> automatically sets the value to 1.
OFFSET	Specifies the offset for inserting records in an array. This option is used with the <code>INSERTSIZE</code> option. If an array includes four names and the <code>OFFSET</code> is set to 2, the second name will be the starting point for inserting data. The <code>Dbset()</code> function specifying this option will occur before an <code>Exec()</code> function. If the <code>OFFSET</code> is specified as a range larger than the <code>MAXARRSIZE</code> , a database error will occur.

WAITRES Specifies whether or not to wait for resources. If this option is set to **TRUE**, the script will wait indefinitely for requested information from the database. If it is set to **FALSE** and the script does not receive the requested information, the script will receive an error that the resource was not available. Script execution will then either continue or exit based on the condition set in `Dberror()`. The default value of this option is **TRUE**.

DEFER Specifies whether or not to defer the `Parse()` statement. If this option is set to **TRUE**, a deferred parse will be performed when the script encounters the `Parse()` function. If the option is set to **FALSE**, a normal `Parse()` is executed. The default value of this option is **FALSE**.

Normally, the SQL statement is sent over to the database when the script encounters `Parse()` and processed to ensure it is semantically correct. The SQL statement is stored to wait for the `Exec()` call that actually executes it. If **DEFER** is set to true, the SQL statement is not sent over to the database until the script encounters an operation that requires input from the database such as `Exec()` or `Describe()`.

A complete list of `Dbset()` options that may appear in the script file is listed below with brief descriptions. This list is divided into those options that set integer values and those options that specify **TRUE** or **FALSE**. The list also designates the options that apply specifically to the database environment as a whole, to the log on structure, or to the cursor.

Integer Value

<u>Option</u>	<u>Description</u>
FETCHSIZE	number of rows to be fetched
MAXARRSIZE	maximum number of rows to be fetched or inserted
INSERTSIZE	number of rows to be inserted
OFFSET	row number where to begin inserting
MAXCONNECT	maximum number of connections in an environment
PACKETSIZE	maximum packet size on logon connection
ROWCOUNT	maximum number of rows to return
TEXTSIZE	limit on size of text or image data
ISOLATIONLEVEL	transaction isolation level
AUTHOFF	turns specified authorization off

TRUE or FALSE

Database environment only

Database environment or logon structure

Logon structure

Copyright PERFORMIX, Inc. © 1995

ARITHABORT	what to do on arithmetic errors
ARITHIGNORE	what to do on arithmetic div. by 0
CURCLOSETRAN	close all cursors on end transactions
NONSTANDSQL	flag non standard sql
FORCEPLAN	force plan or not
FORMATONLY	only send format for data
GETDATA	returns extra information on every sql statement
NOROWCOUNT	row count
PARSEONLY	only check syntax, do not execute
QUOTEDIDENT	all double quotes signify identifiers
PARSETREES	returns parse resolution trees
SHOWPLAN	generates a description of procedure plan
IOSTATS	return internal io statistics
TIMESTATS	return time statistics
IGNORETRUNC	ignore truncation errors
<i>Cursor</i>	
AUTOCOMMIT	commit on every exec
WAITRES	wait for resources instead of error
SPECIAL	sybase version of cursor
HIDDEN_KEYS	expose hidden keys

7.1.3.2 Data

The `Data()` function is used with database operations for inserting or updating data. It specifies the data to be input to the database by listing the data for each input variable specified with `Bind()`. Therefore, it will follow all `Bind()` functions and will occur before `Exec()`. This function is inserted into the script when data is specified to be input to the database.

The `str` parameter is pipe-delimited and lists the data for each input variable that was defined with `Bind()`. The data listed in this `str` parameter will be listed in the same order as it was specified in the `Bind()` functions.

The `Dbset()` option `INSERTSIZE` applies to this function in that the number of `Data()` functions inserted will correspond to the value of `INSERTSIZE`.

Example:

```
Parse(CUR1, SELECT ENAME, EMPJOB, WHERE ENAME=:name...);  
...  
Bind(CUR1, "NAME", STRING, 5);  
...  
Data(CUR1, "SMITH");
```

In this example, SMITH is the data to be entered into the database for the cursor, CUR1.

In another example, a SQL statement may include the following:

```
...EMPNO, ENAME, EMPJOB WHERE EMPNO=:empno, ENAME=:name,  
EMPJOB=:empjob...
```

A Data() line that refers to this SQL statement may look like the following:

```
Data(CUR1, "123|Smith|typist")
```

123 would correspond to empno, Smith would correspond to ename, and typist would correspond to empjob.

Because all users would not make the same entries to the database, you may edit this function to vary the data that is inserted into the database during multi-user emulations for a more realistic load. You can edit the Data() functions in a set of scripts so that each emulated user inserts a different record. For example, if a Parse() statement contains WHERE NAME = SMITH, and the Data() line is Data(CUR1, "SMITH"), you could change this Data() line to Data(CUR1, "JONES") or Data(CUR1, "ADAMS") for each emulated user.

7.1.4 Editing Database Functions

The EMPOWER/CS database functions translate client application and database interaction and are inserted into the script as the associated actions occur. This process ensures that during script execution a load is generated on the SUT as close to the real interactions as possible. Because the EMPOWER/CS database functions are not user-defined but are inserted into your script based on how the client application interacts with the SUT, you generally should not attempt to edit these functions or remove them from the script.

Most of the database functions should not be edited because they are inserted during Capture in a way specific to the application running from the PC to the SUT. If you change these functions from when they were captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

However, simple edits can be made to some of the `Dbset()` options and more complex edits can be made for the `Data()` and `Parse()` functions to enhance multi-user emulations.

Because all users would not make the same entries to the database, you may edit the `Data()` and `Parse()` functions to vary the data that is inserted into the database during multi-user emulations for a more realistic load. You can edit the `Data()` functions in a set of scripts so that each emulated user inserts a different record.

The `Dbset()` option, `FETCHSIZE`, can be changed to see how a larger or smaller `FETCHSIZE` value affects the performance of your SUT during script execution. Please note that the `FETCHSIZE` value can not be larger than the `MAXARRSIZE` value.

As with all C language programs, your script file should include comments to indicate the structure of the script. Comments also are useful for interpreting the log file.

Comments are entered either during Capture or when editing your script file. They are enclosed by the `/*` and `*/` characters and may span multiple lines in a script, but may not be nested.

The following is a list of the EMPOWER/CS data types that may be listed in the database script functions, `Bind()` and `Define()`. Format lists how the data types are represented in the script as C language data types. This format does not necessarily represent how the data type is converted and stored on the database.

Type	Format
NUMBER	Internal database binary
DECIMAL	Internal database binary

Money and Date Data Types

Type	Format	Example
MONEY	dollars.cents	103.45
LONGMONEY	dollars.cents	103.4500
DATE	day/month/year hour:minute:seconds	02/10/95 10:06:35
LONGDATE	day/month/year hour:minute:seconds.milliseconds	02/10/95 10:06:35.23

The format for the DATE data type must be 17 characters in length. The format for the LONGDATE data type must be 20 characters in length.

Numeric Data Types

<u>Type</u>	<u>Format</u>
BYTE	char
SHORT	short
INT	int
UINT	unsigned int
DOUBLE	double
FLOAT	float
BOOL	char

For the remaining data types, the format specifies the maximum length for the character array.

Non null-terminated characters

<u>Type</u>	<u>Format</u>
LONGCHAR	char[2 ³¹ -5]
CHAR	char[2000]
MEDCHAR	char[65535]
SHORTCHAR	char[255]

Null-terminated strings

<u>Type</u>	<u>Format</u>
LONGSTRING	char[2 ³¹ -1]
STRING	char[2001-1]
SHORTSTRING	char[256-1]

Binary data

<u>Type</u>	<u>Format</u>
LONGBINARY	char[2 ³¹ -1]
SHORTBINARY	char[256]
BINARY	char[65535]
IMAGE	char[2 ³¹ -5]

Encrypted data

<u>Type</u>	<u>Format</u>
SECURITY1	char[]
SECURITY2	char[2000]

Arguments also may be used to control the execution of functions in the script. You may have designed the script to contain functions that test different parts of your application. The argument can specify which functions execute at what time and in what sequence.

EMPOWER/CS will define these argument variables automatically as follows:

These variables can be used subsequently in the script. The number of arguments (and therefore the number of elements in the array) is limited only by your UNIX script driver which generally imposes no practical limitation on script execution.

When a script is executed, the following variables are defined:

```
argv[0] = the executable script name
argv[1] = the log file name
```

Arguments specified after the log file name in the script command are stored as `argv[2]`, `argv[3]`, and so on. You must specify a log file name if you want to specify other arguments. If you specify other arguments without specifying the log file, EMPOWER/CS will use the first argument as the log file name, thus creating an error.

7.3.2 Argument Examples

The following example illustrates a script execution:

```
$ script1 -d vagrant log1 1 3 wendy password1
```

`script1` is the name of the binary script created by the Csccl tool. The option `-d` and `vagrant` specify that the activities will be displayed on the PC named `vagrant`. `log1` is the name of the log file that will be created when the script is executed. The parameters `1`, `3`, `wendy`, and `password1` are arguments that will be accessed during script execution. `1` and `3` will be used as arguments for EMPOWER/CS functions; `wendy` will be used as the log in ID; and `password1` will be the emulated user's password.

These arguments are accessed by the script through the variables `argc` and `argv`. The values of the variables are:

```
argc = 6
argv[0] = script1
argv[1] = log1.1
argv[2] = 1
argv[3] = 3
argv[4] = wendy
argv[5] = password1
```



```
Thinkuniform(atoi(argv[2]), atoi(argv[3]));`  
Seed(atoi(argv[3]));
```

The variable array `argv[]` contains character string variables. Since the EMPOWER/CS functions `Thinkuniform()` and `Seed()` take integer arguments rather than character strings, you must use the C library function `atoi()` to convert the character strings to integers.

```
Thinkuniform(1.000, 3.000)
Seed(3.000)
```

EMPOWER/CS scripts are C language programs and therefore, you may use standard C variables in the script source file. All variables used in a script must be defined at the beginning of the script with the exception of `argc` and `argv`, which are defined automatically as described in the previous section.

Variables are defined with the `int`, `char`, `float`, and `double` C language statements. These statements define variables as follows:

<code>int</code>	-	integer
<code>char</code>	-	character
<code>float</code>	-	single precision floating point
<code>double</code>	-	double precision floating point

These statements may be used to define several variables with the variables separated by commas. They also are used to define an array. The following examples show definitions of each variable type:

```
int i, j, k;
int i_array[10];
char c;
char product[20];
char *product_ptr;
float weight;
double iq;
```

In these examples, the variable array `i_array` contains ten integers, accessed as `i_array[0]` through `i_array[9]`. The `product` array can contain 20 characters. The variable `*product_ptr` is used to point to a character string.

After variables are defined, they may be assigned values:

```
i = 10;
i_array[3] = 17
c = 'a';
iq = 17.325;
product_ptr = "072148";
```

You also may define a variable and give it a value in one statement:

```
int i = 11
```

To use a variable in a script, simply reference it by name.

7.5 Editing Your Scripts

After you have created a script with Capture, you likely will want to edit the script to change or add script functions. Of course, the changes you make to your scripts are driven by your testing goals but EMPOWER/CS scripts generally are edited to achieve the exact emulation environment desired. Because EMPOWER/CS scripts are actually C language programs, they can be edited to include branching and looping. Special EMPOWER/CS functions can be added to your scripts for advanced control of script execution. The following sections describe EMPOWER/CS functions that can be inserted into your scripts only by editing, and also describe methods for branching and looping.

7.5.1 Recording Messages

The `Log()` function records a message in the executed script's log file. `Log()` functions are similar to comments in that they provide help in following the structure of a script. The parameter of the `Log()` function must be a character string.

The `Note()` function is used to place a note in the "Note" column of the EMPOWER/CS Monitor View 5 during script execution. The parameter of `Note()` is a character string which is saved in the log file and subsequently displayed by Monitor.

The `Inote()` function specifies a message that also will appear in the "Note" column of Monitor View 5. The parameter of `Inote()` is an integer that will be displayed by Monitor.

You can insert these functions into your script as needed when you edit your script file.

7.5.2 File Input/Output Functions

Creating scripts that read input files on the UNIX script driver is a common practice during load testing. Such input files may contain names, addresses, phone numbers, etc., that are used for emulating database queries and updates. EMPOWER/CS allows you to have a separate input file for each emulated user, or multiple users can share a single input file. Using a single input file is often more convenient since you do not have to maintain many files for large tests.

EMPOWER/CS provides a convenient way for scripts to read from an input file, ensuring that data for each script is unique. The File I/O functions are used in scripts to read and write files which is useful for load tests requiring interaction with data files on the UNIX script driver machine. These capabilities also are useful in simplifying complex scripts such as database entry scripts. The EMPOWER/CS file input/output functions allow you to read data from a file, send data from the file to the SUT, receive data from the SUT, and write those data to a file. These functions simplify the C language statements that would need to be added to accomplish the same thing. You can insert File Input/Output functions when editing your script.

The environment variable `E_FIOPATH` can be used to specify the directory in which the files to be accessed reside. A file must be opened before it can be accessed with the file input/output functions. `Fioopen` may be used to open a file. Also, the `Fioreadline`, `Fioreadfield`, `Fioreadchar`, and `Fiowritechar` functions automatically open the file before reading from or writing to it. The function `Fioshare` is used to identify a file that is to be shared. If a file contains NULL characters, an error will occur when the file is read by an input/output function.

Three global variables are used for file input/output. They are defined automatically as follows:

```
unsigned char *FIOBUFFER
int FIOLEN
int FIOBUFFERSZ
```


The variable `FIOBUFFER` is a pointer to the characters read from the file. This variable often is used when sending data read from a file to the SUT. The variable `FIOLEN` is the number of valid characters in `FIOBUFFER`. If the value of `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file was reached. The variable `FIOBUFFERSZ` is the maximum size of the data that can be read at one time. The default value of `FIOBUFFERSZ` is 512 characters. If the value of `FIOBUFFERSZ` is redefined in a script, it must be redefined before any file input/output functions that reference the file are encountered.

The file input/output functions are:

<code>Fioshare</code>	<code>Fioreadline</code>	<code>Fioskipline</code>	<code>Fioseek</code>
<code>Fiounshare</code>	<code>Fioreadfield</code>	<code>Fioskipfield</code>	<code>Fioautorewind</code>
<code>Fioopen</code>	<code>Fioreadfields</code>	<code>Fioskipchar</code>	<code>Fiorewind</code>
<code>Fioclose</code>	<code>Fioreadchar</code>		
<code>Fiodelimiter</code>	<code>Fiowritechar</code>		

These functions are described in the following sections.

7.5.2.1 Fioshare

The syntax for this function is:

```
Fioshare(filename)
```

The `Fioshare()` function identifies a file that is to be shared. It must be called before any other File I/O functions are called to reference the same file. `Fioshare` in a script presumes execution of the `fioshare` command at the UNIX script driver's shell prompt. The `fioshare` command creates a global variable that contains the offset for the next byte to be read from a shared file. The value of the variable (offset) remains between tests, so you can continue to read an input file from the point left by the previous test. This saving of the offset is useful in tests that corrupt a database on the server. The ability to avoid the same transactions means you can avoid restoring the database before every test. You must execute the `fioshare` command if you want to resume reading from the beginning of the

7.5.2.3 Fioopen

The syntax for this function is:

Fioopen(filename, mode)

The `Fioopen()` function opens the file `filename`. The parameter `mode` specifies how the file is opened. If `mode` is `"r"`, the file is opened at the beginning for reading only. If `mode` is `"w"`, the file is truncated or created for writing only. If `mode` is `"a"`, the file is opened at the end for writing only. If `mode` is `"r+"`, the file is opened at the beginning for reading and writing. If `mode` is `"w+"`, the file is truncated or created for reading or writing. If `mode` is `"a+"`, the file is opened at the

To remove the global variable offset, you must use the `gv_rm` shell command to remove the variable named in the `gv_stat` output listed above. For example:

The `Fioopen()` function opens the file `filename`. The parameter `mode` specifies how the file is opened. If `mode` is `"r"`, the file is opened at the beginning for reading only. If `mode` is `"w"`, the file is truncated or created for writing only. If `mode` is `"a"`, the file is opened at the end for writing only. If `mode` is `"r+"`, the file is opened at the beginning for reading and writing. If `mode` is `"w+"`, the file is truncated or created for reading or writing. If `mode` is `"a+"`, the file is opened at the

end for reading and writing. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.4 Fioclose

The syntax for this function is:

Fioclose(filename)

The `Fioclose()` function closes the file `filename`. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.5 Fiodelimiter

The syntax for this function is:

```
Fiodelimiter(filename, delimiters)
```

The `Fieldelimiter()` function defines the field delimiters for the file `filename`. The default is `"\t\n"` and `"\n"` is always a delimiter ("`\t`" is tab and `"\n"` is new line or newline). If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.6 Fioreadline

The syntax for this function is:

```
Fioreadline(filename)
```


7.5.2.9 Fioreadchar

The syntax for this function is:

```
Fioreadchar(filename, n)
```

The `Fioreadchar()` function reads `n` bytes from the file `filename`. If the file is not currently open (see `Fioopen`), it is opened by `Fioreadchar()`. A pointer to `FIOBUFFER` is returned. If `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file was reached.

7.5.2.10 Fiowritechar

The syntax for this function is:

```
Fiowritechar(filename, buf, n)
```

This function writes `n` bytes from the buffer, `buf`, to the file `filename`. If the file is not currently open (see `Fioopen`), it is created or truncated and opened for reading and writing automatically. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.11 Fioskipline

The syntax for this function is:

```
Fioskipline(filename, n)
```

The `Fioskipline()` function skips forward `n` lines in the file `filename`. If the file is not currently open (see `Fioopen`), it is opened automatically. The variables `FIOBUFFER` and `FIOLEN` are updated with the last line read. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.12 Fioskipfield

The syntax for this function is:

```
Fioskipfield(filename, n)
```

The `Fioskipfield()` function skips forward `n` fields in the file `filename`. The fields are separated by the delimiter (see `Fiodelimiter`). If the file is not currently open (see `Fioopen`), it is opened automatically. The variables `FIOBUFFER` and `FIOLEN` are updated with the last field read. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.13 Fioskipchar

The syntax for this function is:

Fioskipchar(filename, n)

The `Fioskipchar()` function skips forward `n` characters in the file `filename`. If the file is not currently open (see `Fioopen`), it is opened automatically. The variables `FIOBUFFER` and `FIOLEN` are updated with the last characters read (the number of characters used to update `FIOBUFFER` and `FIOLEN` is defined by the variable `FIOBUFFERSZ`). If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.14 Fioseek

The syntax for this function is:

```
Fioseek(filename, offset)
```

The `Fioseek()` function sets the file pointer to a specific byte in the file. The next byte read or written will occur at `offset` bytes from the beginning of the file. If the value of `offset` is equal to `FIOEND`, the seek will occur to the end of the file. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.15 Fiorewind

The syntax for this function is:

```
Fiorewind(filename)
```

The `Fiorewind()` function rewinds the file pointer to the beginning of the file for the file `filename`. If the function is successful, zero is returned. If an error occurs, -1 is returned.

7.5.2.16 Fioautorewind

The syntax for this function is:

Fioautorewind(filename)

The `Fioautorewind()` function automatically rewinds the file pointer to the beginning of the file specified in `filename`. whenever the end-of-file is encountered. This function is useful if multiple scripts read data from one file.

7.5.2.17 File Input/Output Function Examples

The following example illustrates using the functions `Fioreadfield()` and `Fioreadfields()` to read one or more fields from a file. `Fiodelimiter()` is used to specify that the field delimiter in the file `inputfile` is a comma.


```
char last[20];
char first[20];

...

Fioopen("inputfile", "r");
Fiodelimiter("inputfile", ",", "");
Fioreadfield("inputfile");
Type("%s", FIOBUFFER);

LeftButtonPress(360,211);

AppWait(0.06);
WindowRcv("SfSfPt");

Fioreadfields("inputfile", 2, last, first);
Type("%s", last);

LeftButtonPress(357,252);

AppWait(0.06);
WindowRcv("SfSfPt");

Type("%s", first);
```

The following is a portion of the file `inputfile`:

```
312890463
doe,jane
294028190
smith,john
```


The following examples show you how to use the rewind functions to reuse data in an input file. Notice that `Fioautorewind()` saves you some programming time because you do not have to check to see if you are at the end of the file:

```
Fioautorewind("info");
Fioreadline("info");
Type("%s", FIOBUFFER);
```

OR

```
Fioreadline("info");
if (FIOLEN== -1){
  Fiorewind("info");
  Fioreadline("info");
}
Type("%s", FIOBUFFER);
```

7.5.3 Pacing Functions

EMPOWER/CS allows you to insert three functions into your script file for controlling the pace of the script. These pacing functions cause the script to pause long enough to make the script send transactions to the SUT at a predetermined throughput. Typically used in a loop, these functions may be nested to permit controlled throughput of transactions within a larger transaction.

Each of the pacing functions accepts an argument naming the pace and defining the speed of the pace.

`Paceconstant()` causes transactions to be submitted at a constant throughput. The second argument to `Paceconstant()` defines the number of seconds that the script should take since the last call to `Paceconstant()`. The first call to a pacing function does not delay; it is used as a starting point for the subsequent calls. For

this reason, the first call to a pacing function is often made with arguments of 0 to define the speed.

`Paceuniform()` and `Pacetne()` control throughput but do not do so at a constant rate. The functions have an average throughput that will be maintained, but submission of transactions occurs at frequencies other than that defined by a constant distribution.

`Paceuniform()` accepts two arguments to identify the speed of the pace - a minimum delay and a maximum delay. The average of the two will be maintained during sustained execution of the script. Each time `Paceuniform()` is executed, it will delay for a number of seconds since the last execution, where the number is taken from a uniform distribution between the minimum and maximum values. For example, if `Paceuniform("query", 8.0, 12.0)` is used in a script, a sequence of 9, 11, 8, 10, and 12 second delays is possible (assuming the `query` has a response time of zero seconds.) The average of the delays is still ten seconds. `Paceuniform()` will select the values for delay accurate to 1/100th of a second, so 9.27 seconds is a possible value.

`Pacetne()` accepts three arguments to identify the speed of the pace - a minimum, average and a maximum delay. The average will be maintained during sustained execution of the script. Each time `Pacetne()` is executed, it will delay for a number of seconds since the last execution, where the number is taken from a truncated, negative exponential distribution defined by the three values. In a typical such distribution, the average is relatively close to the minimum. For example, if `Pacetne("query", 7.0, 10.0, 20.0)` is used in a script, a sequence of 7, 8, 10, and 15 second delays is possible. The average of the delays is still ten seconds. `Pacetne()` will select the values for delay accurate to 1/10th of a second, so 9.2 seconds is a possible value.

Five EMPOWER/CS functions exist that allow you to manipulate script variables. They are `CmpVar()`, `GetVar()`, `GetIntVar()`, `SetVar()`, `SetIntVar()`. These functions allow you to test, update, read, and compare variables.

You may insert the `CmpVar()` function into your script to compare a variable from a SQL statement to a specified value. The syntax for this function is:

```
CmpVar(curnum, var, value);
```

This function determines if the specified variable, `var`, in the current row is equal to the value specified in the `value` parameter. The address of the variable must be passed to `CmpVar()`.

This function could be used for fetching records until a certain value is returned. An example of this function follows:

```
Parse(CUR1, "select ename from employee_table");

Define(CUR1, "1", STRING, 50);

Exec(CUR1);

Dbset(CUR1, FETCHSIZE, 1);

while (CmpVar(CUR1, "1", "Smith") != 0){
    Fetch(CUR1);
    GetNextRow(CUR1);
}
```

You can insert the `GetVar()` or `GetIntVar()` functions into your script to return the current value of a specified variable (either the name or the position) in the SQL statement. `GetVar()` is used for string variables and `GetIntVar()` is used for integer variables. These functions should be inserted into the script after the `Fetch()` and `GetNextRow()` functions.

The following example demonstrates using the `GetVar()` function within a script:

```
char *empname, *empno;

...

Parse(CUR1, "select ename, empno from employee_table");

Define(CUR1, "1", STRING, 50);
Define(CUR1, "2", INT, 4);

Exec(CUR1);

Dbset(CUR1, FETCHSIZE, 1);
while (Fetch(CUR1) != 0){
    GetNextRow(CUR1);
    empname=GetVar(CUR1, "1");
    printf("empname is %s\n", empname);
    empno=GetVar(CUR1, "2");
    printf("empno is %d\n", *(int *)empno);
}
```

The `SetVar()` and `SetIntVar()` functions assign a new value to a specified variable. The syntax for these functions follows:

```
SetVar(curnum, var, value);
SetIntVar(curnum, var, value);
```

The new value for the variable, `var`, is assigned in the parameter value. These functions could be inserted into the script before an `Exec()` function or after the `GetNextRow()` or `GetVar()` functions.

The following example demonstrates using SetIntVar() in a script:

```
int empno,deptno,i;

...

Parse(CUR1, "select ename from employee_table where empno=:empno
and deptno=:deptno");

Bindp(CUR1, 1, INT, 4);      /* pos 1 is :empno */
Bindp(CUR1, 2, INT, 4);      /* pos 2 is :deptno */

Data(CUR1, "23|20");

empno=GetIntVar(CUR1, "1");
deptno=GetIntVar(CUR1, "2");

/* keep executing above parse statement changing the empno
everytime */
for (i=0;i<5;i++){
    empno=empno+i;
    SetIntVar(CUR1, "2", empno);

    Exec(CUR1);

    if (Fetch(CUR1) != 0)
        printf("empno %d is in deptno %d\n", empno, deptno);
    else
        printf("empno %d is not in deptno %d\n", empno,
deptno);
}
```

7.5.5 Looping

EMPOWER/CS scripts may be modified to contain loops controlled by the standard C language `for` and `while` statements. You can use a loop to emulate hour long activity instead of having to capture activity for an hour. The `for` statement executes a set of EMPOWER/CS functions a fixed number of times. The `while` statement executes a set of EMPOWER/CS functions as long as a given condition is satisfied.

You must insert loops very carefully if you plan to execute scripts in both Non-Display and Display modes. You must be sure to encompass all relevant functions in the loop for Non-Display and Display script execution. For example, you must include all relevant button presses within the loop so that the correct sequence of buttons will be activated during script execution in Display mode.

For your convenience, you should insert a comment or function in the script to mark where you wish to begin and/or end looping.

7.5.5.1 Looping with the For Statement

Executing a loop a certain number of times in a script is accomplished with the `for` statement. The syntax of the `for` statement is:

```
for (initial; condition; step)
{
    /* body of the loop */
}
```

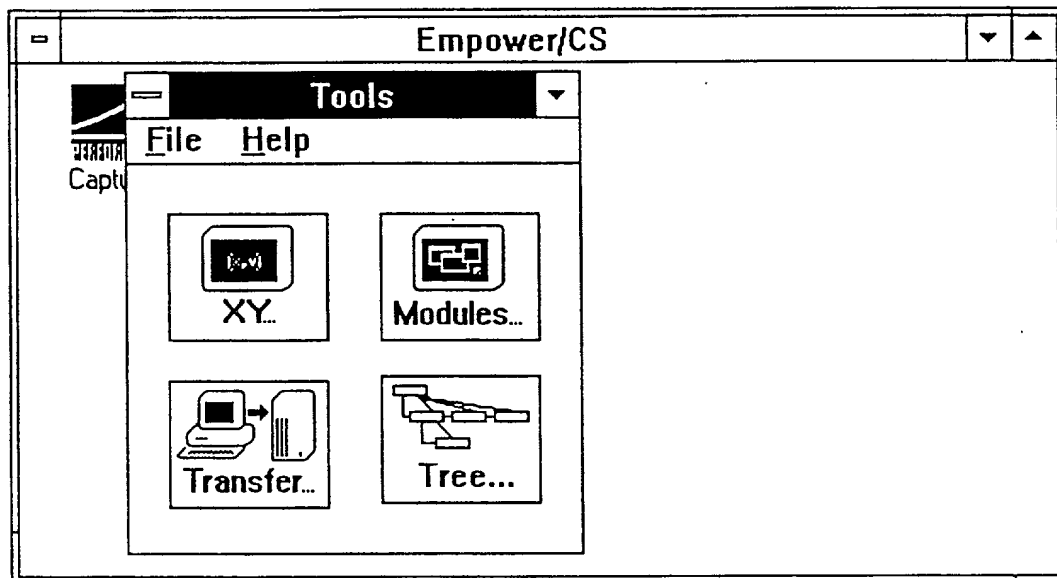
The parameter `initial` generally contains the initial assignment of a looping variable. The parameter `condition` specifies the condition under which looping should continue. This condition is evaluated every time the loop is about to be entered. The parameter `step` generally specifies the increment or decrement of the looping variable. Note that the body of the loop is contained between braces, { }.


```
Fetch(CUR1);
while (GetNextRow(CUR1)!=NOMOREROWS);
```

8.0 EMPOWER/CS Tools

EMPOWER/CS offers four Windows features under the Tools icon. These Tools are XY, Modules, Transfer, and Tree.

When you activate Tools in the EMPOWER/CS window, the following window opens:



8.1 XY

If you wish to add or change mouse button events (i.e., `LeftButtonDown()`, `LeftButtonUp()`, `RightDblPress()`, etc.) in the script file, you will need to know the on screen xy coordinates of the locations that are to be activated. The XY tool helps you to locate on screen xy coordinates of the PC mouse.

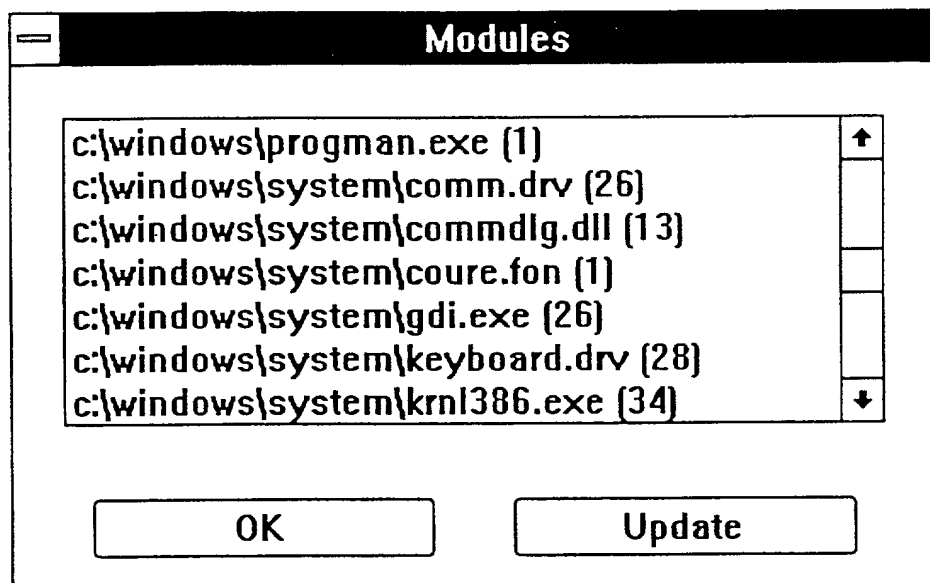
Activate the **XY** pushbutton. The following window appears displaying the current xy coordinates of your mouse:

The **Trace** option of this tool follows your mouse movements listing the on screen xy coordinates as the mouse moves. If you de-select **Trace**, you can type a number in the left field that will move the mouse to that x position on screen. If you tab to the next field, you can enter a y position that will move the mouse to that y position. As you type each number, the mouse will move to the specified position.

Select **Exit** from the **File** menu to close this tool.

8.2 Modules

Select the **Modules** button to activate the following window:



This tool alphabetically lists all the processes currently running in Windows. The number listed in parentheses indicates how many instances of the process are running. **Update** updates all process information.

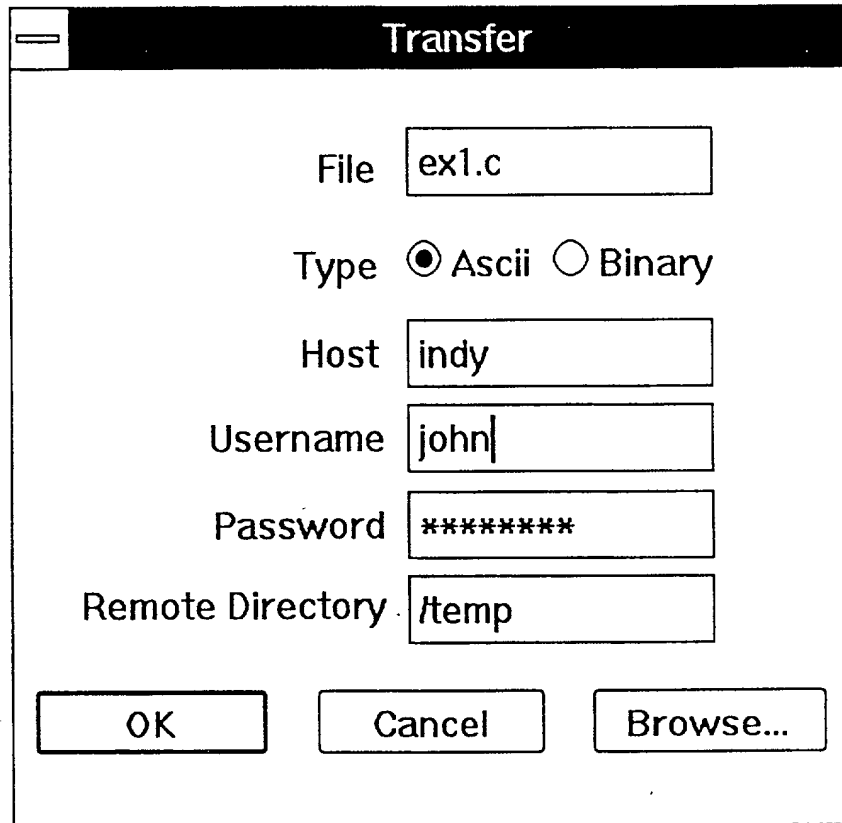
The Modules tool can be used for debugging. For example, if you encountered an error when trying to Capture client/server activity, you could use Modules to verify that `winsock.dll` is installed on your PC.

Select **OK** to exit this tool.

8.3 Transfer

The Transfer tool allows you to transfer files manually from the PC to the UNIX script driver.

Select the Transfer pushbutton in the Tools window. The following window will open:



A screenshot of a 'Transfer' dialog box. The title bar is black with the word 'Transfer' in white. The dialog box has a white background. It contains several labeled text input fields: 'File' with 'ex1.c', 'Type' with radio buttons for 'Ascii' (selected) and 'Binary', 'Host' with 'indy', 'Username' with 'john', 'Password' with '*****', and 'Remote Directory' with '/temp'. At the bottom are three buttons: 'OK', 'Cancel', and 'Browse...'.

Transfer	
File	ex1.c
Type	<input checked="" type="radio"/> Ascii <input type="radio"/> Binary
Host	indy
Username	john
Password	*****
Remote Directory	/temp
OK Cancel Browse...	

Enter the complete name of the script or data file you wish to transfer and specify in what form you wish to transfer the file, either as **ASCII** or **Binary**.

In DOS files, an end-of-line includes a carriage return and a linefeed whereas in UNIX an end-of-line requires only a linefeed. Transferring a file as **ASCII** translates the carriage return/linefeed combinations in a DOS file to the single linefeed required in UNIX.

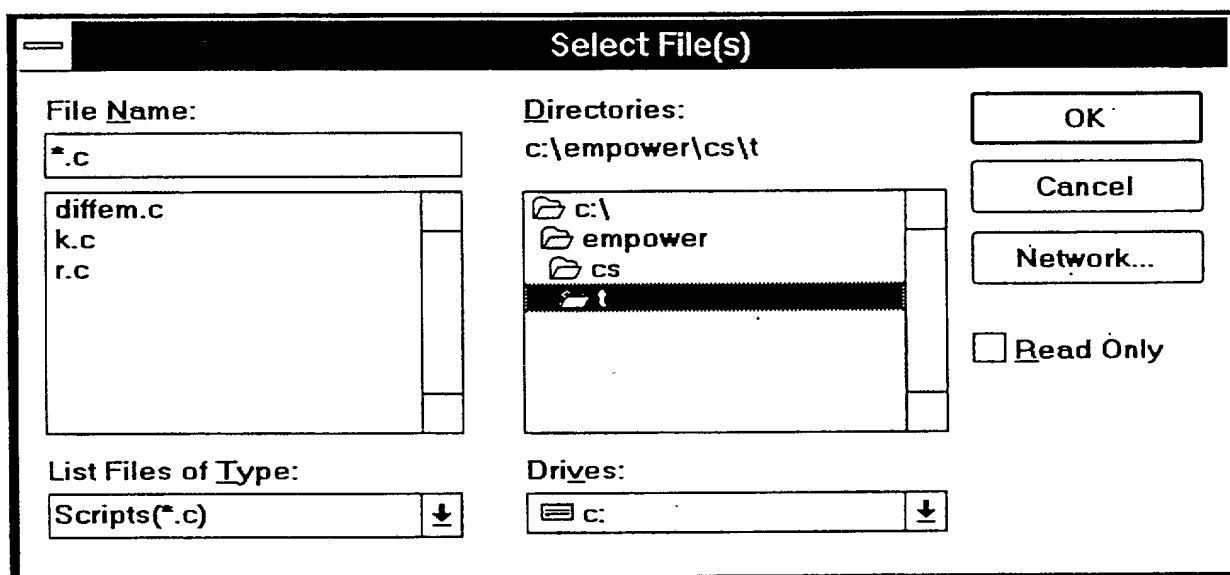
If you specify **Binary**, the file is transferred without character translations. Therefore, you should specify **Binary** for transferring files in which data can not be altered and all characters are required for the file (i.e., images).

If large amounts of data (i.e., images, large text files, etc.) are input to the database during Capture, such data is inserted into separate data files. If you are transferring

a script that has data files, you must transfer the associated data files separately in Binary mode. Scripts should be transferred as ASCII files.

If you need to locate a file, select the **Browse** pushbutton in the **Transfer** window to browse all available directories.


The following **Browse** window will open:



In this window, you may specify or search through all available directories, files, and disk drives. You may select multiple files from a specific directory. When you have located all needed files, choose **OK** or **Cancel** as appropriate to return to the **Transfer** window.

After specifying a file, you should enter the **Host** name (the UNIX driver), the appropriate **Username** and **Password**, and the **Remote Directory** (on the UNIX driver) where you wish to transfer the script file.

In the **Transfer** window, select **OK** to transfer the specified file to the UNIX driver machine.

A screenshot of a Windows-style warning dialog box. The title bar is black with the word "Warning" in white. The main area is white and contains a circular icon with a black exclamation mark on the left. To the right of the icon, the text "Script 'ex1.c' exists on 'indy', overwrite?" is displayed. At the bottom, there are two buttons: "OK" and "Cancel". The "OK" button has a dotted border, while the "Cancel" button has a solid border.

Warning

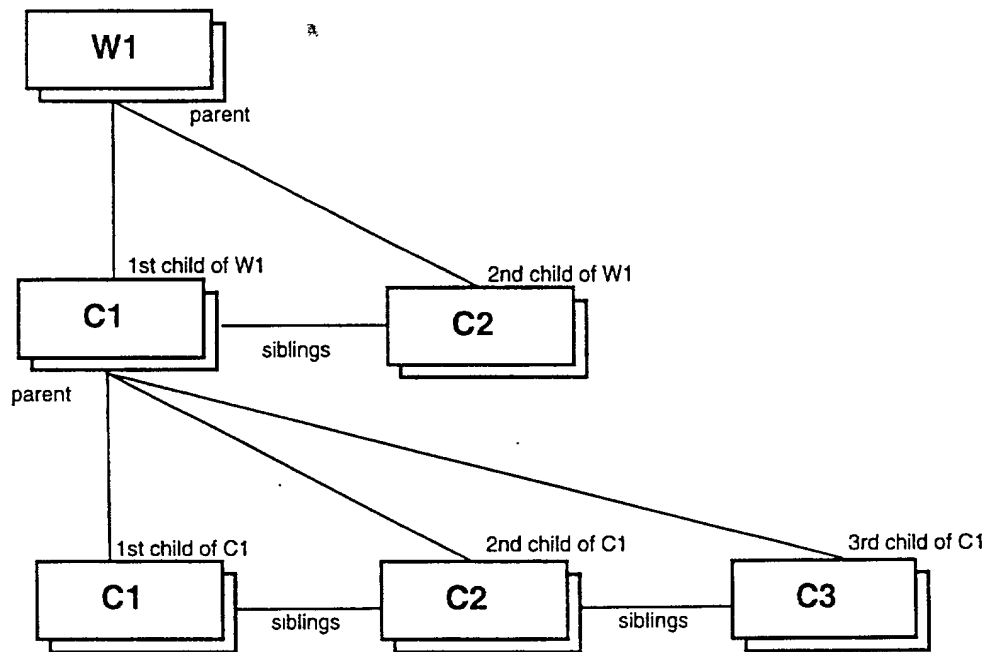
! Script 'ex1.c' exists on 'indy', overwrite?

OK Cancel

EMPOWER/CS-V1.0.1

accessible from a parent window, they are children of that parent window and siblings of each other.

The following diagram demonstrates this structure:



The Tree tool lists a tree structure from right to left, from child to parent, where the right-most item is the name of the button or window you selected. If one of the windows or the pushbutton has no title, something like "#c1" will be listed to indicate the window or pushbutton is a certain numbered child of the preceding parent window.

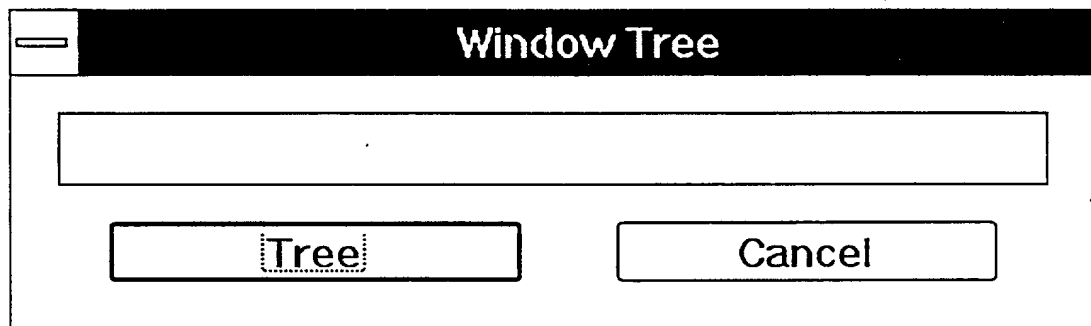
Note: In some rare cases, applications are designed where a window requires its sibling window to also be its parent in which case something like "#s1" may be listed in the structure.

A tree structure from the above diagram could look like: W1|#c1|#c2. In this example, #c2 is the second child (#c1) of the window W1.

In the following tree structure, **Tools|WindowTree|Cancel**, **Cancel** is the button contained in the window **WindowTree** which is a child of the **Tools** window.

8.4.2 Using the Tree Tool

Select the **Tree** button from the **Tools** window which opens the following window:



Activate the **Tree** button. The mouse pointer will turn into a large arrow.

With this new pointer, select the button for which you wish to list the Tree structure. This tree structure is listed in a particular hierarchy as explained above in Section 8.4.1.

As an example, obtain the Tree structure for the **Cancel** button, by selecting **Tree**. Then, select **Cancel** in the **Window Tree** window as shown below:

The following structure will appear in the **Window Tree** window:

Tools|Window Tree|Cancel

When you have obtained all needed tree structures, select the **Cancel** button or **Close** from the **Window Tree** window's **System** menu.

APPENDIX 3

© Copyright 1995
Pure Software, Inc.
All Rights Reserved

04697994.102600

1.0	Introduction	1-1
1.1	Multi-User Testing Tools.....	1-1
1.2	User's Guide Organization.....	1-2
1.3	User's Guide Conventions.....	1-4
<hr/>		
2.0	Mix	2-1
2.1	The Mix Table	2-1
2.2	Mix Syntax	2-4
2.2.1	Batch Mode.....	2-5
2.2.2	Turning Off the Mix Display.....	2-6
2.2.3	The Mix Log.....	2-6
2.2.4	Specifying Ports for Mix daemons.....	2-7
2.3	Mix Commands	2-7
2.3.1	?.....	2-8
2.3.2	!.....	2-8
2.3.3	At.....	2-9
2.3.4	Close.....	2-10
2.3.5	Connect.....	2-10
2.3.6	Exec.....	2-10
2.3.7	Get.....	2-11
2.3.8	Help.....	2-11
2.3.9	Kill.....	2-12
2.3.10	Pause.....	2-13
2.3.11	Put.....	2-13
2.3.12	Quit.....	2-13
2.3.13	Rcd.....	2-14
2.3.14	Rcmd.....	2-14
2.3.15	Resume.....	2-14
2.3.16	Set.....	2-15

3.0	Extract	3-1
3.1	Response Time	3-1
3.2	Defining Transactions	3-2
3.3	Time Stamp Categories.....	3-3
3.4	Accounting for Midnight	3-5
3.5	Extract Syntax.....	3-5
	3.5.1 Selecting Timestamps to Extract.....	3-6
	3.5.2 Options to Handle Timeouts.....	3-7
	3.5.3 Redirecting Extract Output.....	3-10
	3.5.4 Log File Specification.....	3-11
3.6	Sample Extract Execution.....	3-13

EMPOWER/CS V1.0.1

5.0	Draw	5-1
5.1	Batch Mode Execution	5-3
5.2	Interactive Mode Execution	5-5
5.3	Bar Chart Format.....	5-7
	5.3.1 Header.....	5-7
	5.3.2 Figure.....	5-8
	5.3.3 Data Table.....	5-10
	5.3.4 Messages.....	5-10
5.4	Draw Syntax.....	5-11
	5.4.1 Default Interactive Mode.....	5-11
	5.4.2 Quick Interactive Mode.....	5-12
	5.4.3 Batch Mode with Specification File.....	5-14
	5.4.4 Batch Mode with Output File Mode.....	5-14

5.5	Specification Syntax	5-14
5.5.1	INPUT.....	5-15
5.5.2	EVENT.....	5-16
5.5.3	X.....	5-16
5.5.4	Y.....	5-17
5.5.5	TITLE.....	5-17
5.5.6	XTITLE.....	5-17
5.5.7	YTITLE.....	5-18
5.5.8	YMIN.....	5-18
5.5.9	YMAX.....	5-19
5.5.10	LEGEND.....	5-19
5.5.11	ORGANIZE.....	5-19
5.5.12	COMMENT.....	5-20

6.0	Monitor	6-1
6.1	Shared Memory.....	6-2
6.2	The Curses Library.....	6-2
6.3	Monitor Syntax	6-4
6.3.1	-.....	6-4
6.3.2	-e.....	6-5
6.3.3	-k.....	6-5
6.3.4	-o.....	6-6
6.3.5	-r.....	6-6
6.3.6	-w.....	6-7
6.3.7	-A.....	6-7
6.3.8	-S.....	6-8
6.3.9	-v.....	6-9
6.3.10	-i.....	6-9
6.3.11	-s.....	6-9
6.4	Starting Monitor	6-10
6.5	Monitor Help Screens	6-12
6.6	First Help Screen Commands.....	6-14
6.6.1	Moving.....	6-14
6.6.2	Scrolling.....	6-15
6.6.3	Searching.....	6-15

6.6.4	Escape Modes.....	6-16
6.6.4.1	Trace Mode.....	6-17
6.6.4.2	Edit Mode.....	6-18
6.6.4.3	View Mode.....	6-18
6.6.4.4	Zoom Mode.....	6-19
6.6.5	Other Commands.....	6-20
6.6.5.1	^.....	6-20
6.6.5.2	^l.....	6-20
6.6.5.3	ni.....	6-20
6.6.5.4	s.....	6-21
6.6.5.5	C.....	6-22
6.6.5.6	d.....	6-22
6.6.5.7	q.....	6-23
6.6.5.8	?.....	6-24
6.7	Second Help Screen Commands.....	6-24
6.7.1	Deleting.....	6-24
6.7.2	Killing.....	6-25
6.7.3	Flushing.....	6-26
6.7.4	Resuming.....	6-26
6.7.5	Suspending.....	6-27
6.7.6	Interrupting.....	6-27
6.7.7	Joining.....	6-28
6.7.7.1	Joining from Monitor.....	6-29
6.7.7.2	Joining from within a Script.....	6-29
6.8	Third Help Screen Commands.....	6-30
6.9	Views.....	6-31
6.9.1	EMPOWER Monitor Views.....	6-32
6.9.1.1	EMPOWER Monitor View 1.....	6-33
6.9.1.2	EMPOWER Monitor View 2.....	6-35
6.9.1.3	EMPOWER Monitor View 3.....	6-36
6.9.1.4	EMPOWER Monitor View 4.....	6-37
6.9.1.5	EMPOWER Monitor View 5.....	6-38
6.9.1.6	EMPOWER Monitor View 6.....	6-39
6.9.1.7	EMPOWER Monitor View 7.....	6-40
6.9.2	Monitor Views for EMPOWER/X.....	6-41
6.9.2.1	EMPOWER/X Monitor View 1.....	6-41
6.9.2.2	EMPOWER/X Monitor View 2.....	6-43
6.9.2.3	EMPOWER/X Monitor View 3.....	6-44

EMPOWER/CS V1.0.1

After you have generated performance reports for your multi-user emulation, Draw will accept one or more reports to produce bar charts that depict relationships among performance results. These relationships can summarize a single multi-user test or a series of multi-user tests in which each test contains a different user level or system configuration.

The Global Variables tool, EMPOWER/GV provides advanced control over multi-user emulations by creating and controlling global variables that are shared among executing scripts. Variables can be defined to direct scripts to terminate gracefully when they run an indefinite process. They can be used as a counter for scripts that must perform a fixed amount of work before terminating. Variables also may be used to synchronize the execution of multiple scripts. The elements of EMPOWER/GV include commands and functions which control access to a variable; read, update, or test values of variables; or control a shared memory segment.

General use information and specific technical reference material for operating the PERFORMIX load testing products are provided in several manuals. Examples have been provided for clarity.

This manual, *Multi-User Testing*, is divided into the following sections:

- Section 1: Provides an introduction to multi-user testing
- Section 2: Describes the Mix tool which combines scripts to develop complete emulations of systems or applications to be tested
- Section 3: Describes the Extract tool which collects performance data from log files of executed scripts
- Section 4: Describes the Report tool which produces statistical reports of results of multi-user performance tests
- Section 5: Describes the Draw tool which graphically displays results of multi-user performance tests
- Section 6: Describes the Monitor tool which displays critical information about scripts during execution. It also can control script execution

Note: This manual is used for the PERFORMIX Inc. products EMPOWER, EMPOWER/X, and EMPOWER/CS. The EMPOWER, EMPOWER/X, and EMPOWER/CS tools for script development are all maintained separately. However, the tools for multi-user testing, as described in this manual, are common among the EMPOWER products. Some commands and situations may apply only to one of the products. These instances are specifically noted within the text.

You may notice different version numbers for any one of the multi-user tools. If you have purchased two or more EMPOWER products, the version number for the multi-user testing tools will correspond to one of the EMPOWER products purchased.

The conventions followed in this User's Guide are listed below:

Regular Font	Used for all regular body text
Mono-spaced Font	Used for all command, function, and file names; for all examples; and, generally, for any computer-generated text
Bold Mono-spaced Font	In examples, represents entries made by the EMPOWER, EMPOWER/X, or EMPOWER/CS user
<code>[-p port]</code>	In command syntaxes, text within brackets represents optional command parameters
<code>gv_stat[name -s]</code>	Vertical lines () separate command parameters
...	Within scripts, the ellipsis marks indicate some script content was left out for brevity
<code>Beginfunction()</code>	Parentheses are included with script functions mentioned in regular body text. For most functions, one or more parameters will be listed in the parentheses
<code>Endscenario()</code>	EMPOWER/CS script functions use initial capitalization
<code>extract</code>	EMPOWER/CS command names use all lower case letters
Mix	When an EMPOWER, EMPOWER/X, or EMPOWER/CS tool is mentioned within regular body text, it is shown in regular font with initial capitalization

The term "SUT" refers to your system under test.

Complete, thorough emulations must stress a system under test (SUT) to verify that it can handle predicted workloads. Therefore, EMPOWER, EMPOWER/X, and EMPOWER/CS must be able to emulate various users operating multiple terminals connected to the SUT (or for EMPOWER/CS, multiple PCs connected to the server).

2.1 The Mix Table

Specifications for a multi-user emulation are provided in a mix table which identifies users and specifies the script each user will emulate. Each line in the table directs the execution of a single script, and the size of the mix table is unlimited. In the following example mix table, we will emulate six users working simultaneously. An example for each EMPOWER product is included.

The first item you enter in a mix table is the script ID which represents the emulated user name. The script ID is used by the Mix and Monitor tools to identify each script individually. Following the user name and comma in each line is the

script execution command which includes the executable script file name and any valid script execution options. These options are fully defined in the Compiled Script Execution sections of the *EMPOWER Script Development*, *EMPOWER/X Script Development*, and *EMPOWER/CS Script Development* manuals.

While executing scripts, Mix will create log files for each script which are named by adding a .1 extension to the script ID. The default logs files for the following example scripts will be user1.1, user2.1, user3.1, and user4.1:

```
user1, script1 telnet:sut
user2, script1 telnet:sut
user3, script2 telnet:sut
user4, script2 telnet:sut
```

If E_PORT equals telnet:sut, you do not need to specify telnet:sut. The above Mix table could look like the following:

```
user1, script1
user2, script1
user3, script2
user4, script2
```

If you want to name the script log files different from the script id, you must type the names after the specified port.

Example:

```
user1, script1 telnet:sut log1
user2, script1 telnet:sut log2
user3, script2 telnet:sut log3
user4, script2 telnet:sut log4
```

If you wish to pass your own arguments to the script, you are required to specify the port and log for each script.

Example:

```
user1, script1 telnet:sut log1 user001 pass001
user2, script1 telnet:sut log2 user002 pass002
user3, script2 telnet:sut log3 user003 pass003
user4, script2 telnet:sut log4 user004 pass004
```

You may want a single emulated user to execute several scripts consecutively, which is achieved by replacing the user name for subsequent scripts with a "+", as shown in the following example. You will need to specify different log names or subsequent scripts will overwrite the default log files of the previous script execution.

```
u1, query telnet:sut log1a
+ report telnet:sut log1b
+ modify telnet:sut log1c
```

In the above example, one ASCII terminal user executes the script called `query`, then the script called `report`, then the script called `modify`. The results are saved in the Mix log files `log1a`, `log1b`, and `log1c`, respectively.

A similar process applies to an X terminal user in the following example:

```
u2, report report log2a
+  query  query  log2b
+  modify modify log2c
```

For a PC user:

```
u3, report log3a
+   query log3b
+   modify log3c
```

To simulate the time a user might spend between different operations (different scripts), you may add a "sleep" period before executing another script.


```
u1, query telnet:sut log1a
+ sleep 5 report telnet:sut log1b
+ sleep 5 modify telnet:sut log1c
```

2.2 Mix Syntax

```
$ mix -
```

EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

Usage:

```
mix [-f mixfile [-d]] [-l log] [-I][[-D] [-p port]]
```

Options:

- f mixfile Runs MIX in batch mode reading commands from mixfile
- d Prevents display of MIX session on screen
- l log Causes the MIX trace to be stored in log
- p port Specifies the port Daemon should listen on

Note:

MIX runs in interactive mode by default.
In interactive mode, type ? to obtain more help.
Use the -f and -d options when running MIX in the background.
By default, the MIX trace is stored in mix.log.
Default MIX Daemon port is 7273.

Examples:

```
mix
mix -f run3user
mix -f run6user -d &
mix -l 3user.log
mix -p 3333
```


Typically, one command file is generated for each configuration to be tested. For example, if you are testing your SUT with one, eight, and thirty-two users, you might create three separate command files called `1user.cmd`, `8user.cmd`, and `32user.cmd` that would look like the following:

Note: Any lines in a batch file that start with "# " are assumed to be comments and are ignored during script execution.

An example Mix log file is shown below:

```
15:38:27.53 mix> use tab
15:38:30.47 mix> set tstart 2
15:38:34.75 mix> start all
15:38:38.06 [user01] started
15:38:40.17 [user02] started
15:38:42.17 [user03] started
15:38:44.17 [user04] started
15:38:44.23 mix> wait
16:35:26.36 user01 terminated (3/4) running
16:35:26.36 user02 terminated (2/4) running
16:35:28.44 user03 terminated (1/4) running
16:35:28.44 user04 terminated (0/4) running
16:35:28.45 no more scripts running
16:35:28.45 mix> quit
```

2.2.4 Specifying Ports for Mix daemons

Relevant only to distributed emulations with Mix, the `-p` option of `mix` specifies the port that Mix daemons should listen on. The default Mix daemon port is 7273. Refer to Section 2.5.2 for a more detailed explanation of distributed emulations with Mix.

2.3 Mix Commands

The Mix tool allows you to control script execution in the mix table by providing several Mix commands. Mix is a command interpreter, so it displays its own prompt on the UNIX script driver when executed interactively. The Mix commands may be entered at this prompt or may be contained in command files for batch execution.

2.3.1 ?

The ? command displays the Mix command help screen:

```
$ mix
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

mix> ?

command      syntax              description
-----
at            at n cmd             run MIX command at n seconds
close        close               forget about entries in last script table used
connect      connect host ...    connect to MIX daemon on host(s)
exec         exec commandfile    execute file of MIX commands
get          get host file ...   copy file(s) from host
kill         kill all | id ...   terminate execution of script(s)
pause        pause n              sleep for n seconds
put          put host file ...   copy file(s) to host
quit         quit                exit MIX
exit         quit                exit MIX
rcd          rcd host dir        change directory on host
rcmd         rcmd host cmd       run cmd on host
resume       resume all | id ...  continue execution of suspended scripts
set          set [variable value] initialize a MIX variable
signal       signal all | id ...  send signal to script(s)
start        start all | id ...   begin execution of script(s)
stat         stat [-l] [id]      display script status
table        table [-a] name fmt  make a table using format
time         time                display current time of day
trap         trap command        run command if !cmd exits with error
use          use table           read script table
wait         wait                pause until all scripts complete
mix>
```

2.3.2 !

The ! command executes a shell command from within Mix on the UNIX driver machine. Upon completion of the shell command, control returns to Mix and Mix displays its prompt.

Examples:

```
mix> !date
Thu Oct  6 15:49:35 EDT 1995
mix> !ps
  PID TT STAT  TIME COMMAND
23917 p2 IW   0:00 -sh (sh)
25302 p2 S    0:00 mix
25311 p2 S    0:00 sh -c ps
25312 p2 R    0:00 ps
mix>
```

2.3.3 At

The **at** command is used to execute another Mix command at a specified time. The syntax of the **at** command is shown below:

```
at n command
```

The integer **n** represents the time to execute the command, and **command** is the Mix command to be executed. The time is specified in seconds from the beginning of the Mix session. The **at** command is useful for executing Mix in batch mode. For example, the following command will execute the Mix command **start all** one minute (60 seconds) after the Mix session has started:

```
mix> at 60 start all
```

If the **at** command executes after the specified time has expired, a warning will be displayed and the command will execute. The following example Mix session shows the use of the **at** command 30 seconds after the session began:

```
mix> at 5 time
warning: at time has passed
time: 12:11:05
```

2.3.4 Close

The `close` command ends the use of one mix table before proceeding to another mix table. A `close` command must be used before a second `use` command is entered. Refer to section 2.3.23 for a description of the `use` command.

Example:

```
mix> use table1
mix> start all
mix> wait
mix> close
mix> use table2
```

2.3.5 Connect

The `connect` command is used only when running Mix as a daemon on multiple UNIX driver machines. This command connects the specified remote Mix daemon(s) to the central Mix session. Refer to Sections 2.5.2 and 2.5.3 for a more detailed explanation of distributed emulations with Mix.

2.3.6 Exec

The `exec` command, generally used in interactive mode, directs Mix to retrieve and execute a series of commands that are stored in a separate file. This capability is helpful for executing a frequently used set of commands without constantly retyping them. Notice that the user controlling Mix types only two commands to start the test.


```

mix> exec s4
mix> use 4usertab
mix> set tstart 2
mix> set tresume 20
mix> set logdir ./logs
mix> !date
Thu Oct 6 15:59:24 EDT 1995
mix> start all

```

The `get` command is used when running Mix as a daemon on multiple UNIX script driver machines. This command copies a specified file(s) from a specified remote UNIX driver machine to the primary UNIX driver machine. Refer to Section 2.5.3 for a more detailed explanation.

The `help` command displays the help menu. This command performs the same action as `"?"`.

Example:

```
mix> use table1
mix> start all
mix> pause 600
mix> kill all
mix> wait
mix> quit
```

```
mix> kill user[01-07]
```

```
mix> kill user[02,04,12]
```

```
$ ls chap[1-50]
```

2.3.10 Pause

The `pause` command causes Mix to delay for a period of seconds. `Pause` often is used in command files when an emulation must execute for a specified time before terminating. `Pause` can be used when a large number of scripts require time to log in and suspend themselves before all scripts can be resumed.

Example:

```
mix> use table9
mix> start all
mix> pause 20
mix> resume all
mix> wait
mix> quit
```

2.3.11 Put

The `put` command is used when running Mix as a daemon on multiple UNIX script driver machines. This command copies a specified file(s) from the primary UNIX driver machine to the specified remote machine. Refer to Section 2.5.3 for a more detailed explanation of distributed emulations with Mix.

2.3.12 Quit

The `quit` command causes Mix to terminate. If scripts are executing, Mix will ask for verification. If you decide to quit while scripts are executing, the scripts will continue until they terminate themselves. If Mix is executed in batch mode, the `quit` command will not ask for confirmation. (*Note:* You can also enter `q` or `exit` which function exactly as `quit`.)


```
mix> quit
$
```

The `red` command is used when running Mix as a daemon on multiple UNIX script driver machines. This command causes the Mix daemon on the specified remote UNIX driver machine to change to a specified directory. Refer to Section 2.5.3 for a more detailed explanation of distributed emulations with Mix.

The `rcmd` command is used when running Mix as a daemon on multiple UNIX script driver machines. This command executes a specified command on a specified UNIX driver machine. Refer to Section 2.5.3 for a more detailed explanation.

The `resume` command causes `Mix` to resume suspended scripts. The `resume` command must be followed by a list of identifiers or the key word `all` which resumes all scripts.

A script execution is suspended when it executes the `Suspend()` function. The `Suspend()` function is useful for multi-user emulations that require all users to be logged into the SUT before transactions can be executed or response times can be measured.

Example:

The `resume` command allows the specification of a range. For example, to resume the scripts `user01`, `user02`, ..., `user07`, the following command is used:

To specify multiple scripts that are not continuous, use the command specification shown in the following example:

2.3.16 Set

TSTART controls the delay between starting scripts. TRESUME controls the delay when resuming scripts. T SIGNAL controls the delay between signaling scripts. LOGDIR defines the directory where log files of scripts to be executed are stored.

TABLE indicates the mix table that will be used for the session. CONTINUE has two values (on or off) that indicate whether or not continuation scripts will be executed. (Continuation scripts begin with "+" in the Mix table.) All variables are recognized in lower case letters, for example, `tstart` and `tresume`.

The default values of `TSTART` and `TRESUME` are five seconds and `TSIGNAL` is zero seconds. If you prefer, you can specify these values as floating point numbers which allows you to start or resume scripts a half-second apart, one-and-a-quarter seconds apart, etc. The default value of `LOGDIR` is the current directory. TABLE has no default, and the default for CONTINUE is "on".

Examples:

```
mix> use 4 usertab
mix> set tstart 2
mix> set tresume 10
mix> set logdir ./logs
mix> set
TSTART 2.00
TRESUME 10.00
TSIGNAL 0.00
LOGDIR ./logs
TABLE 4usertab
CONTINUE on
mix>
```

2.3.17 Signal

The Mix `signal` command sends a signal to a script to control execution. For example, a `signal` command can be used with the file input/output routines to read type rate values from another file. The script must include a `Signal()` function to specify an action to take when the script receives the signal. If the script does not include a `Signal()` function, the `signal` command will be ignored.

The syntax of the signal command is:

```
signal scriptid [...]
```

scriptid specifies a script where the signal should be sent. The parameter of scriptid may be "all", in which case the signal is sent to all scripts.

The following example shows a portion of a script that uses the `Signal()` function. When the Mix user enters a signal command, the type rate will be read from a file:

```
typechange()
{
  int i;
  Fiorewind("/tmp/typerate"); /* go to beginning of file */
  Fioreadline("/tmp/typerate"); /* read line in */
  if(FIOLEN > 0) { /* if no error */
    i=atoi(FIOBUFFER); /* convert value to int */
    if(i > 0)
      Typerate(i); /* set the new type rate */
    else
      Signal(IGNORE); /* ignore further signals */
  }
}

Empower(argc, argv)
int argc;
char **argv;
{
  Thinkuniform(1,2.5);
  Timeout(300,EXIT);
  Signal(typechange); /* set the Signal handler */
  Beginscenario("shell");
  /* ...
  rest of script
  ... */
  Endscenario("shell");
}
```

When this script executes, the type rate initially is set to the default of zero characters per second. The first time the Mix signal command is entered at the UNIX script driver, the value in the file `/tmp/typerate` is read and, if the value is not zero, the type rate is set to the specified value. This process repeats every time

the `signal` command is entered until the value read from the file is zero. At that point, all subsequent signals are ignored.

2.3.18 Start

The `start` command executes one or more scripts. To begin executing one or more scripts, enter the `start` command followed by the script ids from the mix table. To start all scripts in the mix table, enter the `start` command followed by the key word `all`.

When starting multiple scripts, each script will begin after a delay determined by the TSTART variable.

Examples:

```
mix> set TSTART 10
mix> start user1 user2 user3
[user1] started
[user2] started
[user3] started
```

The `start` command allows specification of a range. For example, to start the scripts `user01`, `user02`, ..., `user07`, enter the following command:

```
mix> start user[01-07]
```

To specify multiple scripts that are not continuous, such as user02, user04, and user12, use the following command format:

```
mix> start user[02,04,12]
```


Example:

```
mix> stat
user1  user2*  user3
```

Examples:

```

mix> stat -l
script_id  process_id  sleep  script
-----
      u01           -1      0  example1 telnet:sut user1
      u02           23      0  example2 telnet:sut user2
      u03           -1      0  example2 telnet:sut user3
summary: 1/3 running

```


The `Mix` command `table` builds a mix table from within `Mix`, which is useful for building large mix tables. The syntax of the `table` command is:

```
table filename arg1 arg2 ... argn
```

The parameter `filename` specifies the name of the new mix table, and the arguments `arg1` through `argn` are entries for the Mix table columns.

The `table` command is most useful when used with a range specification, as shown in the following example:

```
mix> table mix.tab user[01-50], script tty[01-50] user[01-50].1
```

This command would produce the following mix table:

```
user01, script tty01 user01.1
user02, script tty02 user02.1
user03, script tty03 user03.1
...
user50, script tty50 user50.1
```

The `time` command displays the current time.

Example:

```
mix> time
time: 16:09:51
```


[illegible]

The Mix shell escape command (!) is used to escape Mix temporarily to execute shell commands. The Mix trap command specifies an action to be taken if the executed shell command fails, or returns a non-zero exit status.

The syntax of the `trap` command is:

```
trap cmd
```

The parameter `cmd` specifies the action to be taken which can be "exit", "continue", or any valid Mix command (including another shell escape command). For example, when a shell escape command returns a non-zero exit status, Mix will exit if the value of `cmd` is "exit"; or it will continue executing if the value is "continue". The default trap condition is "continue".

The `trap` command can halt execution of Mix, or it can test and reset any function of the emulation environment. This command is useful for executing Mix in batch mode when a shell escape command in the Mix batch file is critical to the execution of the test.

If the shell escape command fails during batch execution, you may not notice an error or be able to end the Mix session before your test is flawed. For example, if the test executes after the shell escape command fails, the false test start could update a database incorrectly which must be rebuilt prior to executing a successful test. By using the trap command set at "exit", you can avoid such an inconvenience.

In the following example *Mix* session, the trap condition is set to "continue," a shell escape command returns an error, and the *Mix* session continues:

```

mix> trap continue
mix> use mixtab
mix> set tstart 2
mix> !setupfile
sh: setupfile: not found
mix> start all

```


Now the same example will be executed with the trap condition set to "exit":

```
mix> trap exit
mix> use mixtab
mix> set tstart 2
mix> !setupfile
sh: setupfile: not found
$
```

Toggling the trap condition between "continue" and "exit" is sometimes useful during a Mix batch execution. Certain shell escape commands may be critical to test execution and if they malfunction, you should exit the test to make appropriate changes. Other shell escape commands may not be important and the test could continue even if the command did not execute. The following example Mix batch file demonstrates this situation:

```
use mixtab
set tstart 2
set LOGDIR logs
!date
trap exit
!critical_cmd
trap continue
!unimportant_cmd
start all
wait
quit
```


Example:

```
mix> use table1
mix> start all
```

Example:

```

mix> use table1
mix> start all
[user1] started
[user2] started
[user3] started
mix> wait
user1 terminated (2/3 running)
user2 terminated (1/3 running)
user3 terminated (0/3 running)
no more scripts running
mix>

```

2.4 Sample Mix Session

In the following example, two emulated users will run one script each. User1 will execute the script `example1` and user2 will execute the script `example2`. The mix table, `table1`, will be used by Mix, and the default log file `mix.log` will be created. During Mix execution, we will instruct Mix to use `table1`, to display script status, to start all scripts, to wait until all scripts have completed, and to quit.

An example of `table1` follows:

```
user1, example1 -n 1 -d serverhost:0
user2, example2 -n 2
```

The example Mix session is presented below:

```
$ mix
Mix: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

mix> use table1
mix> stat -l
script_id  process_id  sleep  script
-----
user1      -1           0      example1 -n 1 -d serverhost:0
user2      -1           0      example2 -n 2
summary: 0/2 running
mix> start all
[user1] started
[user2] started
mix> wait
user1 terminated (1/2 running)
user2 terminated (0/2 running)
no more scripts running
mix> quit
$
```

Extract and Report typically are executed after a multi-user emulation. The following example demonstrates the easiest method for executing the Extract and Report commands after the two-user test:

```
$ extract *.1
$ report
```


After executing the Extract and Report commands, your report can be found in the file `report.STD`.

Sections 3.0 Extract and 4.0 Report fully explain how to operate these multi-user tools.

2.5 Distributed Emulations with Mix

Some load tests require multiple UNIX driver machines to emulate large numbers of users. The mix session on your primary UNIX script driver can connect to and control Mix sessions on additional driver machines. The additional mix sessions act as "daemons" receiving instructions from the primary UNIX script driver's mix session.

Mixd, the tool used on run-time (or remote) driver machines, communicates with the master Mix session on the primary UNIX script driver machine to control scripts. The Mix daemons and master Mix session communicate via TCP/IP network connections. All Mix sessions must connect to and communicate through the same TCP port number. The default port number is 7273.

2.5.1 Installing a Run-Time License

After purchasing the Run-Time option, you will need to install the software on each additional UNIX driver machine. To install a Run-Time license, install your EMPOWER software from the distribution media onto each run-time machine. (Clearly, the run-time machine must be binary compatible with the primary UNIX script driver. Refer to Section 3.0 Installation in the Empower Script Development manual for complete installation instructions.) Change to the Install directory created by reading the EMPOWER tape or diskettes. Run `./emininstall`. Follow the instructions given by Emininstall for contacting Performix customer support to obtain

When you have your installation password, re-run `./emininstall` and type the password when requested. Emininstall will install the binaries that you read off the tape or diskettes. Only `mixd`, the `gv` commands, and `mon`, (or `xmon` or `csmon` as appropriate) are executable on a Run-Time Licensed machine. You must execute all other tools or commands from the primary UNIX script driver machine.

After installing Mixd on all of your run-time UNIX driver machines, you may begin a Mix session on a machine in three ways. The first method is simply to type `mixd` at the run-time machine. The Mix daemon will begin and will run in the background, waiting for instructions from the primary UNIX script driver's Mix session.

The `connect` command attempts to connect to the Mix daemon on the remote UNIX script driver. Whenever a connection is attempted that is not accepted immediately (i.e., if `mixd` is not running), the "`mixdaemons`" file will look for a line containing a command to start the Mix daemon. This command then will execute and the connection should be successful.

An example "mixdaemons" file follows:

```
crosby, rsh crosby /usr/empower/bin/mixd
gemini, rsh gemini /usr/empower/bin/mixd
kitfox, rsh kitfox /usr/empower/bin/mixd
```

The third and most advanced method for starting a Mix daemon involves the `inetd` network process daemon. We recommend that you become proficient with the first two methods for running multiple UNIX script drivers before proceeding with the following method.

A Mix daemon will start automatically whenever the master Mix session connects to the run-time machine through the specified TCP port. If you use this method, you must first specify the communications port to be used by adding the following line to the `/etc/services` file (or local variation) on each machine that will run a Mix daemon:

```
mixd 7273/tcp
```

You may change the TCP port number to any number greater than 1024, and this number must be used for all Mix daemons. The default port number may be overridden with the `-p` option of the Mix or Mixd commands (follow the `-p` with the appropriate port number). This option is necessary only if port number 7273 already is used in the `/etc/services` file of any of the UNIX driver machines.

You must add the following line to the `/etc/inetd.conf` file (or local variation) so that the `inet` daemon runs the correct command when the master Mix session connects to the specified port:

```
mixd\t stream\t tcp\t nowait\t root\t /u/empower/bin/mixd/mixd -I
```

In this line, `\t` represents tab characters. You should replace the directory `/u/empower/bin` with the directory and bin that includes the EMPOWER executables. To ensure that the updated file `/etc/inetd.conf` is used, you must reboot the machine or a `SIGHUP` signal must be sent to the `inetd` process which tells `inetd` to re-read the `inetd.conf` file.

Our example Mix table follows:

```
user01@hosta, ./script telnet:hostc
user02@hostb, ./script telnet:hostc
user03, ./script telnet:hostc
user04@hosta, ./script telnet:hostc
user05@hostb, ./script telnet:hostc
user06, ./script telnet:hostc
```

This table lists entries for starting scripts on two remote UNIX driver machines and on the primary UNIX script driver. (*Note:* Including "." in script names may be necessary so that the Mix daemon can find the script.)

The mix.cmd file follows:

```
# the "use <table>" command will automatically
# connect to all hosts named in the table, but I
# prefer to make the connections explicitly
connect hosta
connect hostb
use table

# remove all old logs in current directory
!rm *.l

# compile the script
!scc script

# change directory to the test directory on hosta and hostb
rkd hosta /usr/testing/test1
rkd hostb /user/testusr/test1

# remove all old logs in the test directories on hosta and hostb
rcmd hosta rm *.l
rcmd hostb rm *.l

# copy the script binary to hosta and hostb
put hosta script
put hostb script
```

(continued on following page...)


```
# eminstall the binaries
# eminstall needs to know the value of the EMPOWER variable on the
# run-time machines
rcmd hosta EMPOWER=/usr/empower $EMPOWER/Install/eminstall script
rcmd hostb EMPOWER=/usr/empower $EMPOWER/Install/eminstall script

# start the test
start all

# wait for the scripts to finish
wait

# get the log files from scripts back to the master Mix
get hosta *.l
get hostb *.l
# extract the timestamps from the log files
!extract *.l

# report from the extract files
!report

# done
quit
```


[This page intentionally left blank]

Using these ASCII files, Report calculates response times and produces statistical reports. You also can use your own statistical software package to evaluate the time stamp data retrieved by Extract.

On many systems and with many applications, the first printable character of a response often denotes that a transaction is in progress. Messages such as "busy..." or "working..." appear when transactions on the SUT begin. In such cases, defining the end of the response time as the time at which the first character

In EMPOWER/CS scripts, transactions are defined by the `BeginTimer()` and `EndTimer()` functions. The word "timer" is used for these functions instead of "transaction" to avoid confusion between true database transactions and the user-level time stamps recorded in a script. In most cases, the term "timer" may be

Functions contain a set of script interactions. The difference between functions and transactions is how the interactions are time-stamped. While individual interactions in a transaction receive time stamps, an entire function receives only one pair of time stamps. Report will indicate the response time for completing all interactions that make up the function. This report will include any think time or type delays that occurred during the set of script interactions.

3.3 Time Stamp Categories

Used only to name transactions in EMPOWER scripts, the `Nametransaction()`, `Begintransaction()`, and `Endtransaction()` functions do not cause time stamps to be written to the log file. EMPOWER automatically recognizes each transaction (i.e., `Xmit-Rcv` pair) and records one of four types of time stamps in the log file as described below.

Response time for functions and scenarios is calculated as the difference between each time stamp for the Begin and End functions. For example, if a Beginfunction() function executes at 21:28:25.52 and the corresponding Endfunction() function executes at 21:28:25.76, then the response time for this

EMPOWER/CS V1.0.1

Notes:

As described in Section 3.3, four types of time stamps are produced for EMPOWER and EMPOWER/X scripts that mark transaction response time: XT1, XT2, RT1, and RT2. By default, Extract extracts the time stamps XT2 and RT2 from the log file. Response time generally is considered to be the difference between XT2 and RT2, since this amount represents the time required for the system to respond completely to an input but does not include the time required for the user to make the input.

EMPOWER/CS V1.0.1


```
$ extract -r *.1
```



```
1
[diane@joelle] cat example2.F
15:35:33.43 zz:zz:zz:zz "shell_commands"
[diane@joelle] extract -s example2.1
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
```

The following EMPOWER log file was created by the script used in the above examples that executes shell commands:

```
>>>      Port: telnet:localhost
>>>      Log: example2.1
>>>      Date: Thu Oct 13 15:35:24 1994
>>>      Command: example2 -d telnet:localhost
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

>>>      Beginsource("example2")
>>>      22 Set(CDELAY)
>>>      23 Typerate(5.00)
>>>      25 Thinkuniform(1.000,2.500)
>>>      26 Seed(7072)
>>>      27 Timeout(30,CONTINUE)
>>>      28 Unset(NOTIFY)

>>>      29 Signal(USER-DEF)
>>>      32 Term(ZOOM, VT100|LINES24|AUTOWRAP)
>>>      34 Rcv(": ")

>>>      RT1 15:35:24.62
SunOS UNIX (joelle)

login:
>>>      RT2 15:35:24.63
>>>      Think 1.499
>>>      XT1 15:35:25.65
>>>      36 Xmit("empower^M")
>>>      XT2 15:35:27.25
>>>      37 Rcv(":")
>>>      RT1 15:35:27.73
```

(continued on following page ...)


```
empower
Password:
>>>      RT2 15:35:27.83
>>>      Think 2.333
>>>      XT1 15:35:29.84
>>>      39 Xmit("WickleOne^M")
>>>      XT2 15:35:31.89
>>>      40 Rcv("] ")
>>>      RT1 15:35:32.03

Last login: Thu Oct 13 15:34:45 from LOCALHOST.perfor
SunOS Release 4.1.2 (GENERIC) #2: Wed Oct 23 10:52:58 PDT 1991

MODEM USERS:  att1:tty22 dialout at up to 2400.
               att1:tty24 dialout at up to 9600.
               joelle:ttya dialin at up to 9600 (703-760-9241).

120MB tape      /dev/rmt0
3 1/2 floppy    /dev/fd0          (fdformat to floppy)

run openwin if you want OpenWindows
inc: no mail to incorporate
[empower@joelle]
>>>      RT2 15:35:33.43
>>>      44 Beginscenario("example2") 15:35:33.43
>>>      46 Beginfunction("shell_commands") 15:35:33.43
>>>      Think 1.893
>>>      XT1 15:35:34.44
>>>      51 Xmit("date^M")
>>>      XT2 15:35:35.47
>>>      52 Rcv("% ")
>>>      RT1 15:35:35.53
date
Thu Oct 13 15:35:35 EDT 1994
[empower@joelle]
>>>      Timeout 15:36:05.67
>>>      57 Endfunction("shell_commands") 15:36:05.68
>>>      Think 1.220

>>>      XT1 15:36:06.70
>>>      58 Xmit("exit^M")
>>>      XT2 15:36:07.76
```

(continued on following page ...)


```
>>> 59 Wait(2)
>>> RT1 15:36:07.78
exit
>>> RT2 15:36:07.78
>>> 61 Endscenario("example2") 15:36:07.92
>>> Endsource()
>>> Closed: telnet port
```

3.5.3 Redirecting Extract Output

When Extract reads a script log file for time stamp data, it places the data in, if appropriate, a transaction file ending with ".T", a function file ending with ".F", and a scenario file ending with ".S". If Extract reads only one log file, the prefix of these files will be the same prefix of the file that Extract read.

Example:

```
$ extract example1
```

The above example would create files called `example1.T`, `example1.S`, and `example1.F`. When Extract scans more than one log file for time stamp data, it places the time stamp data in files called `extract.T`, `extract.S`, and `extract.F`.

In either case, if you want Extract to place the time stamp data in files with a different prefix, you can do so with the `-f` option of the `extract` command. The parameter of the `-f` option is the prefix name of the time stamp data files. The following command will place Extract output in files called `1user.T`, `1user.S`, and `1user.F`:

```
$ extract -f luser example1
```


You also may use the wild card character * to specify multiple files.

```
$ extract example*.1
$ extract user*.1
$ extract *.1
```

Note that Extract will recognize the .l extension and will not try to add an additional extension. In the last example (`extract *.l`, often used in multi-user load tests), Extract will use all files in the current directory that have the .l extension.

When Extract scans multiple files, it will display a number on the UNIX script driver that indicates the remaining number of log files to be scanned. This display is helpful when you have used the command `extract *.1` to scan all files with the .1 extension. If you are running a ten user test, you should see the number "10" appear on the UNIX script driver when Extract executes. As each file is read, new numbers appear, counting down until all files have been read. The following example demonstrates this process after all files have been scanned by Extract:

```
$ extract *.1
Extract: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95

10  9  8  7  6  5  4  3  2  1
$
```

Note: If you execute Extract for a ten user test and the first number displayed is "11," you probably need to delete an old log file.

```
$ extract example1
Extract: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95

1
$
```

The number of log files to be scanned appears below the Extract Copyright statement. In this example the number is 1. Extract places time stamp data in the files `example1.T`, `example1.F`, and `example1.S`. The format of a time stamp is `hh:mm:ss.xx` where `hh` is the hour, `mm` is the minute, `ss` is the second, and `xx` is the hundredth of a second.

Assuming that we had added the appropriate `Nametransaction()` or `Begintransaction()` and `Endtransaction()` functions in the script, the file `example1.T` might look like this:

```
10:56:53.20 10:56:53.48 "date"
10:56:56.40 10:56:52.01 "ls"
```

If not, it would look like this:

```
10:56:53.20 10:56:53.48 " "
```

If we had specified the appropriate `Beginfunction()` or `Endfunction()` functions, `example1.F` might contain the following records:

```
10:56:02.11 10:56:20.56 "query_database"
10:56:20.56 10:56:22.66 "enter_data"
10:56:30.70 10:56:50.08 "vi_session"
10:56:52.21 10:57:02.23 "quit_database"
```


If we hadn't specified `Beginfunction()` or `Endfunction()` functions, the file would be empty.

Because we extracted from only one log file, `example1.s` would contain only two records: the number of log files scanned and time stamps for the beginning and end of the scenario. If we had extracted more logs, each log would have a beginning and end scenario time.

```
1
10:55:54.83 10:57:02.40 "example1"
```


The Report tool reads Extract's output to generate response time and throughput data. This information can be presented in standard transaction, function, and scenario reports or in a Government Services Administration (GSA) compliant report. These reports are stored in two files: one with a .STD extension, and one with a .GSA extension. The standard report is produced by default; the GSA report is generated with the -G option of the `report` command.

The following usage message lists the syntax of the `report` command. You can access this menu by typing the `report` command with a hyphen:

```
$ report -
EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Usage:  report [-BEFSThmdc12345678] [-b t] [-e t] [-f f] [-p n]
        [-u n] [-w n] [-s n] [script]
Options:
-b t      Changes begin time of sample to t
-e t      Changes end time of sample to t
          Default includes events ending in sample
-B        Includes only events starting in sample
-BE       Includes only events starting and ending in sample
-F        Causes a function report to be produced
-G        Causes a GSA report to be produced
-S        Causes a scenario report to be produced
-T        Causes a transaction report to be produced
-h        Suppresses headings
-m        Changes report units to minutes (default is seconds)
-d        Forces within values to be discrete, not cumulative
-c        Suppresses default columns
-[1-8]    Forces default column n to appear
-f f      Changes the output file names to f.GSA and f.STD
-p n      Computes the nth response time percentile
-u n      Changes the number of users to n
-w n      Computes number of events completed within n units
-s n      Changes the size of .STD name fields to n (default: 14)
```

4.1.1 Specifying a Begin and End Time

Report will calculate the duration of each emulated user's session which is used to calculate the system under test's (SUT's) throughput. The duration is the difference between the begin time and the end time for each report.

By default, the begin time is when the first `Beginscenario()` executes and the end time is when the last `Endscenario()` executes. However, Report will calculate statistics for all interactions in the Extract output file, including interactions that occurred before the first `Beginscenario()` execution or after the last `Endscenario()` execution. Therefore, if the default begin and end times are used, activities that occur before the first `Beginscenario()` execution or after the last `Endscenario()` execution will cause erroneous results.

To create reports that calculate data for a specified time other than the default begin and end times, use the `-b` and `-e` options of the `report` command. The `-b` option specifies the report start time; the `-e` option specifies the report end time. The time is specified as hours (hh), hours and minutes (hh:mm), or hours, minutes, and seconds (hh:mm:ss).

You also may use the key words `first` and `last` with the `-b` and `-e` options. The option `-b first` specifies that the begin time is the time of the first `Beginscenario()` execution. The option `-b last` specifies that the begin time is the time of the last `Beginscenario()` execution.

The option `-e first` specifies that the end time is the time of the first `Endscenario()` execution. The option `-e last` specifies that the end time is the time of the last `Endscenario()` execution. The following examples demonstrate the `-b` and `-e` commands:

```
$ report -b 10:20 example1
$ report -b 10:20 -e 11:20 example1
$ report -e last example1
```



```
$ report -Twww 1 2.5 5 example1
```

For the above example, suppose:

- ☐ five events were completed within one second,
- ☐ three more events completed between one and two and a half seconds, and
- ☐ seven more events completed between two and a half and five seconds.

- ☐ one second, five events had completed
- ☐ two and a half seconds, eight events had completed
- ☐ five seconds, fifteen events had completed

- ☐ one second, five events had completed
- ☐ two and a half seconds, three events had completed
- ☐ five seconds, seven events had completed


```
$ report -c -125678 -pp 75 95 -f custom
```


This command creates the following report, custom.STD:

EMPOWER Standard Report

Date: Wed Apr 7 10:52:55 1993
 Start time: 10:41:32
 Stop time: 10:42:26
 Duration: 00:00:54
 Mix: 4 users
 Unit: seconds

Scenario	Total	Finish	Average	Minimum	Maximum	Std-Dev	P75	P95
script1	4	4	48.91	43.79	53.97	4.15	51.86	53.97

Function	Total	Finish	Average	Minimum	Maximum	Std-Dev	P75	P95
word_proc	4	4	3.16	2.62	3.80	0.42	3.19	3.80
db_query	4	4	3.54	2.34	5.71	1.30	3.38	5.71
Overall	8	8	3.36	2.34	5.71	0.99	3.38	5.71

Transaction	Total	Finish	Average	Minimum	Maximum	Std-Dev	P75	P95
date	20	20	0.24	0.10	0.45	0.09	0.28	0.35
loginid	4	4	0.18	0.12	0.33	0.08	0.16	0.33
logout	4	4	0.02	0.01	0.02	0.00	0.02	0.02
ls	20	20	0.28	0.11	0.68	0.13	0.32	0.45
passwd	4	4	5.00	2.58	6.95	1.61	5.80	6.95
pwd	4	4	0.05	0.02	0.13	0.05	0.02	0.13
termtype	4	4	2.42	1.90	3.72	0.76	2.12	3.72
who	20	20	0.26	0.10	0.55	0.11	0.33	0.43
Overall	80	80	0.58	0.01	6.95	1.20	0.33	2.58

4.1.9 Redirecting Report Output

By default, Report places output in a file ending with the suffix, .STD. The prefix is obtained from the Extract output files used to generate reports.


```
$ report -f luser example1
```

4.1.10 Percentile Selection

Response times can be reported by breaking measured transactions into response time percentiles. The `-p` option of the `report` command specifies that response time percentiles should be calculated and reported. You should enter an integer value between one and 100 after the `-p` option. As shown in the following examples, you may set a series of response time percentiles in several ways. These examples specify that a transaction report will generate with the 70th, 80th, 90th, 95th, and 99th percentile response times. Remember that the `-c` option is used to suppress the standard report columns.

```
$ report -c -T -p 70 -p 80 -p 90 -p 95 -p 99 example1
```

```
$ report -c -T -ppppp 70 80 90 95 99 example1
```

```
$ report -c -Tppppp 70 80 90 95 99 example1
```


The example report, `example1.STD` follows:

EMPOWER Standard Report

Date: Mon Oct 17 16:26:59 1994
Start time: 16:23:44
Stop time: 16:25:37
Duration: 00:01:53
Mix: 10 users
Unit: seconds

Transaction	P70	P80	P90	P95	P99
date	0.50	0.69	1.04	1.14	1.65
login	2.04	2.45	2.97	3.22	3.35
logout	0.02	0.02	0.04	0.04	0.04
ls	2.60	2.68	3.06	3.21	3.79
ps	5.78	5.87	6.56	7.72	8.72
who	1.57	1.94	2.46	3.15	3.75
Overall	2.53	3.12	4.84	5.83	7.72

A percentile is defined as a value below which a certain percentage of the observations fall. In this report, 90 percent of all transactions occurred in 4.84 seconds or less, and 90 percent of the date transactions occurred in 1.04 seconds or less.

Note: If the `-p` option is not used, response time percentiles will not be reported.

4.1.11 Specifying Number of Users

By default, Report uses the number of log files as the number of users, which is accurate if each emulated user creates one log file. If the number of emulated users is not the same as the number of log files created, you can change the number of users with the `-u` option of the `report` command. The following command will change the number of users to 32.

```
$ report -u 32 example1
```


The field size in a standard report specifying the transaction, function, or scenario is 14 characters wide by default. With the `-s` option, you can change the field size to a minimum of 11 characters and a maximum of 100 characters. For more column space, use the `-s` option to specify a smaller field size.

The report command's `script` parameter identifies the script that produced the log from which time stamp data was extracted. Report will add a `.S` extension to the file name when searching for the extracted scenario file, a `.F` extension when searching for the extracted function file, and a `.T` extension when searching for the extracted transaction file.

If you do not specify a script, Report looks for the files `extract.S`, `extract.F`, and `extract.T`. Recall that Extract writes to `extract.S`, `extract.F`, and `extract.T` when it extracts from more than one log file.

In addition to configuring reports with command line options, you can configure them with environment variables. Report understands three environment variables.

The first environment variable is E_VENDOR. If E_VENDOR is defined, Report will use its value in the GSA and STD reports as the name of the vendor for whom the report is being prepared. In the example reports shown in the following sections, E_VENDOR is defined as "Your Company Name".

4.3 Sample Report Execution

GSA and Standard reports can be viewed with UNIX commands such as `more`, `cat`, and `pg`. Examples of these reports follow.

4.3.1 The GSA Report

Contents of example1.GSA - Part 1

EMPOWER Scenario Summary Report

Project:	10 User Test
Vendor:	Your Company Name
Date:	Tue Oct 18 14:31:47 1994
Activity:	Your SUT Configuration
Summary for scenario:	Overall
Start time:	16:23:44
Stop time:	16:25:37
Duration:	00:01:53
Number of scenarios attempted:	10
Number of scenarios completed:	10
Completion rate:	0.09 scenarios per second
Median time:	92.75 seconds
Average time:	93.78
Minimum time:	85.68
Maximum time:	101.18
Standard deviation:	4.14
Summary for scenario:	"example1"
Start time:	16:23:44
Stop time:	16:25:37
Duration:	00:01:53
Number of scenarios attempted:	10
Number of scenarios completed:	10
Completion rate:	0.09 scenarios per second
Median time:	92.75 seconds
Average time:	93.78
Minimum time:	85.68
Maximum time:	101.18
Standard deviation:	4.14


```

Project:                               10 User Test
Vendor:                               Your Company Name
Date:                                 Tue Oct 18 14:31:47 1994
Activity:                             Your SUT Configuration
Summary for transaction:               Overall

Start time:                           16:23:44
Stop time:                             16:25:37
Duration:                              00:01:53
Number of transactions attempted:       230
Number of transactions completed:       230
Completion rate:                        2.03 transactions per second

Median time:                           1.29 seconds
Average time:                           1.90
Minimum time:                           0.00
Maximum time:                           8.72
Standard deviation:                     1.89

Summary for transaction:               Unspecified

Start time:                           16:23:44
Stop time:                             16:25:37
Duration:                              00:01:53
Number of transactions attempted:       180
Number of transactions completed:       180
Completion rate:                        1.59 transactions per second

Median time:                           1.58 seconds
Average time:                           2.10
Minimum time:                           0.00
Maximum time:                           8.72
Standard deviation:                     2.03

Summary for transaction:               "who"

Start time:                           16:23:44
Stop time:                             16:25:37
Duration:                              00:01:53
Number of transactions attempted:       50
Number of transactions completed:       50
Completion rate:                        0.44 transactions per second

Median time:                           0.91 seconds
Average time:                           1.18
Minimum time:                           0.28
Maximum time:                           3.75
Standard deviation:                     0.92

```


- Overall Summary—a summary of all occurrences of the interaction
- Unspecified Summary—a summary of unnamed interactions
- Named Interaction Summary—a summary of named interactions

Each GSA report includes a title, the report date, and start and stop times of the extracted data.

The start time for each report is the earliest moment at which a `Beginscenario()` function was executed. The stop time is the moment at which the last `Endscenario()` function completed. The duration is the difference between the start and stop times. The default start and stop times are overridden by the `-b` and `-e` report options as described in section 4.1.1.

The GSA format also displays the number of events attempted and completed. Any interaction that begins during script execution is an attempt, even if it did not end. For example, the execution of a `Beginfunction()` function without the execution of the corresponding `Endfunction()` function is an incomplete attempt.

4.3.1.3 Completion Rate

The completion rate is computed as the number of completed events divided by the duration of the emulation. This completion rate identifies *throughput* of the SUT.

4.3.1.4 Median

The median is the middle value, or the arithmetic mean of two middle values, in a distribution. In this case, median refers to the response time value that half of the interaction response times are greater than and half are less than.

4.3.1.5 Average

The average is the arithmetic mean of a distribution. The average interaction response time is the sum of the response times divided by the number of interactions.

4.3.1.6 Minimum

The minimum is the smallest value in a distribution. In our reports, minimum refers to the minimum interaction response time value.

4.3.1.7 Maximum

The maximum is the largest value in a distribution. In our reports, maximum refers to the maximum interaction response time value.

4.3.1.8 Standard Deviation

Standard deviation is a measure of dispersion in a distribution. It is defined as the square root of the arithmetic average of the squares of deviations from the mean. For our reports, standard deviation is used in conjunction with the average to help determine the amount that each interaction's response time varies from the average.

Note: Using percentiles often is easier for obtaining information on interaction response time distribution.

For more information on these statistical measurements, you can refer to any statistical reference text.

4.3.2 The Standard Report

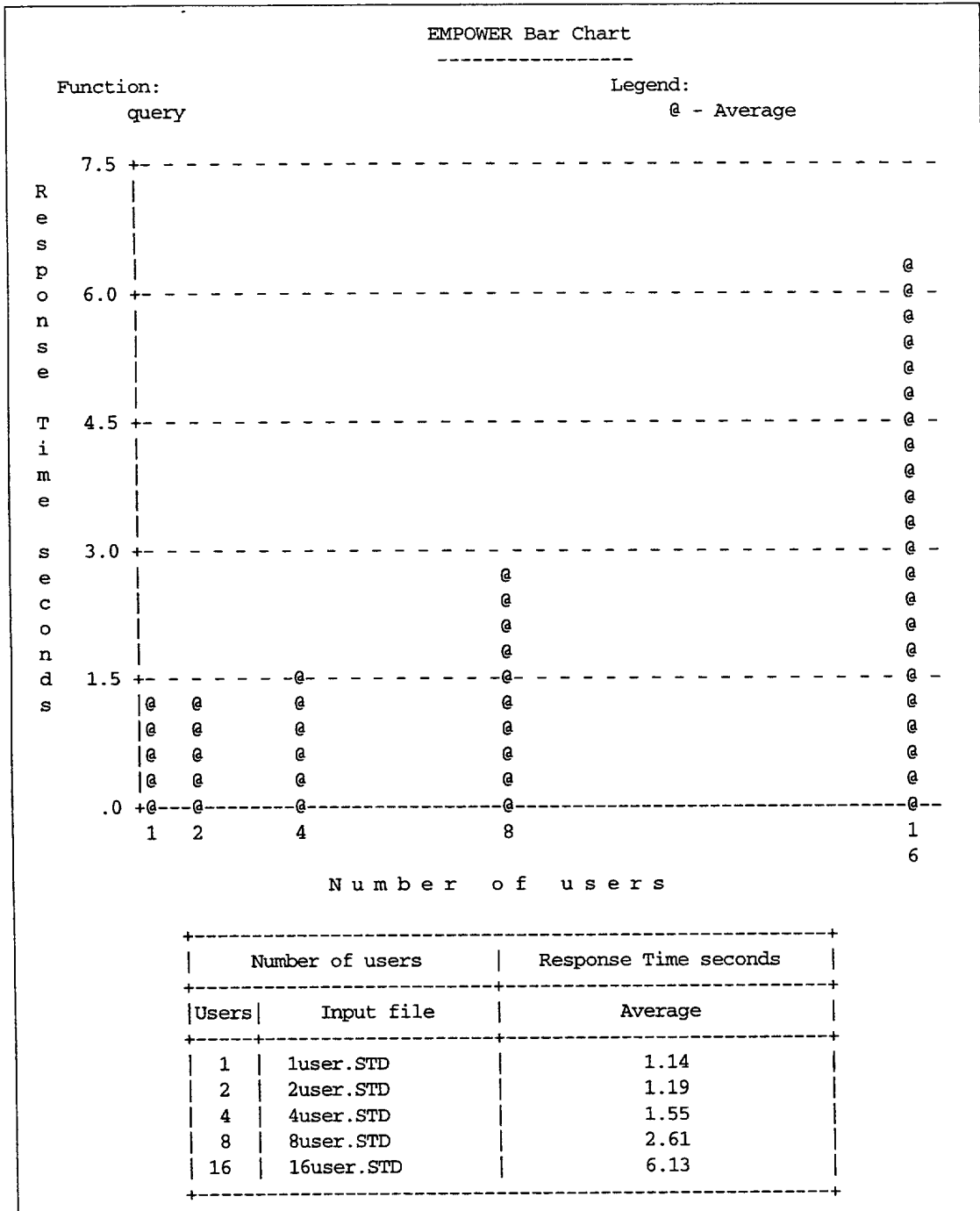
An example of a typical EMPOWER standard report follows:

Draw provides an added dimension to Report results. The output of Report answers such questions about a single user emulation as:

- The output of Draw answers questions about one or more multi-user emulations such as:

- Draw produces bar charts, or histograms, that can be viewed on-line or printed on any system printer. The output of Draw is in ASCII format and contains no printer-specific characters.

Sample Draw Output—Response Time Vs. Number of Users



5.1 Batch Mode Execution

Draw can be executed in batch mode if a file exists that specifies the chart's format. The following example shows a bar chart's specification file that will compare minimum, maximum, and average response time of "query" and "update" transactions during a multi-user run.

```
BEGIN
TITLE = EMPOWER Bar Chart
XTITLE = Transaction
YTITLE = Response Time seconds
INPUT = 16user.STD
EVENT = Transaction
X = query, update
Y = Average, Minimum, Maximum
LEGEND = @, |, *
ORGANIZE = CLUSTERED
YMIN = 0.0
YMAX = 20.
COMMENT = Created for ABC Corp
COMMENT = January 2, 1991
END
```

If this file is called `draw.spec`, then the following command will execute Draw in batch mode. (*Note:* The complete syntax for the `draw` command is discussed in Section 5.4).

```
$ draw -s draw.spec -o draw.out
Draw: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Bar chart(s) drawn.
Draw exited.
$
```

As directed by the specification file, Draw would create a bar chart that places response time on the Y-axis and transaction types on the X-axis. Response times of the "update" transactions are noticeably larger than response times for the "query" transactions.

EMPOWER Bar Chart				
Start time	20:27:01		Legend:	
Stop time	20:32:42		@	- Average
Duration	341.06 seconds			- Minimum
Mix	16 users		*	- Maximum
Input file	16user.STD			

The chart displays response times for two transactions, A and B, across 16 users. The y-axis represents response time in seconds, ranging from 0.0 to 20.0. Transaction A (query) has an average response time of 6.13 seconds, with a minimum of 0.89 and a maximum of 16.35. Transaction B (update) has an average response time of 8.87 seconds, with a minimum of 0.61 and a maximum of 19.98. The chart uses vertical bars to represent the distribution of response times for each transaction, with markers for average, minimum, and maximum values.

Transactions				
Transaction		Response time seconds		
ID	Description	Average	Minimum	Maximum
A	query	6.13	0.89	16.35
B	update	8.87	0.61	19.98

5.2 Interactive Mode Execution

Specifications for bar charts generally are created interactively. To enter interactive mode, change to the working directory where your standard report files are stored and enter the draw command without any options. The working directory must contain some standard report files or Draw will display an error and terminate.

The following example shows an interactive session that will produce the specification and chart shown in section 5.1. In this example, CR indicates pressing the carriage return key:

```
$ draw
Draw: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

Please respond after each => prompt.
Press RETURN to accept a default.
Type quit at any time to exit.

Please select one or more INPUT file names.
Available files in the current directory are:
  1) 16user.STD  2) 1user.STD   3) 2user.STD
  4) 4user.STD   5) 8user.STD
Enter * to select all the files.
=> 1
Please select an EVENT to be charted.
  1) Scenario      2) Function      3) Transaction
Default is 1) Scenario
=> 3
Please select one or more Transactions for the X axis.
Available X axis values are:
  1) Overall      2) Unspecified    3) query      4) update
Default is 1) Overall
=> 3 4
Please select one or more statistics for the Y axis.
Available Y axis values are:
  1) Total        2) Finish        3) Thruput    4) Median
  5) Average      6) Minimum      7) Maximum   8) Std-Dev
Default is 5) Average
=> 5 6 7
Please enter 3 LEGEND characters.
Default characters are @ | *
=> CR
```

(continued on following page...)

(continued on following page...)


```
.
Do you want to create another spec (y/n)?
Default is n) no
=> CR
Do you want to create the bar chart (y/n)?
Default is y) yes
=> CR
Please enter name of output file.
Default is draw.out
=> CR
draw.out exists.
Do you want to append or overwrite (a/o)?
Default is o) overwrite
=> CR
1 bar chart written to draw.out
Draw: normal completion.
```

During this interactive session, a specification file called `draw.spec` was created and the bar chart created was stored in `draw.out`. These two files are identical to the files used in Section 5.1.

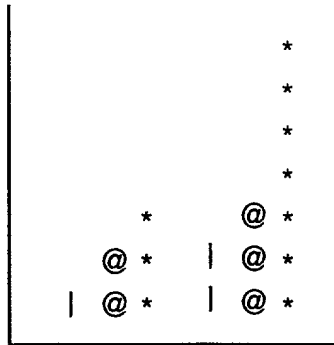
5.3 Bar Chart Format

Every bar chart created by Draw, interactively or in batch mode, is composed of four sections: header, figure, data table, and messages. If it contains 14 or fewer bars or clusters, the entire chart will fit on one standard 8-1/2 x 11 page. If there are more than 14 bars or clusters, the data table will continue to the next page.

5.3.1 Header

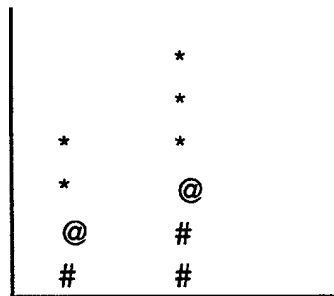
The header is the top portion of the chart. It consists of a centered title at the top, a legend on the right, and chart information on the left. The chart information displayed depends on the number of specified `INPUT` files. If only one `INPUT` file is specified, the start time, stop time, duration, number of users, and name of the `INPUT` file will be displayed. If multiple `INPUT` files are specified, the name of the transaction, scenario, or function that is charted will be displayed.

Example:



If two or three performance measurements are specified for the Y field and the ORGANIZE value is HIDDEN, each bar will contain two or three LEGEND characters. The taller bars will appear hidden behind the shorter bars.

Example:



Due to limited space, the chart occasionally is printed in HIDDEN format even though the ORGANIZE value is CLUSTERED.

When the maximum value of a bar exceeds the value specified by `YMAX`, a `^` character will be printed on the tip of the bar. Similarly, a `v` character will be printed on the horizontal axis at the bottom of the bar when the minimum value of a bar is less than the `YMIN` value.

The location of each bar or cluster depends on the number of specified `INPUT` files. When only one file is specified, the bars or clusters will be distributed evenly on the horizontal axis. If multiple input files are specified, the bars or clusters will be arranged in increasing order according to the number of users identified in each input file. The distance between the bars or clusters will be proportional to the number of users that the bars or clusters represent. In cases where it is impossible

[illegible]

THE UNIVERSITY OF CHICAGO

Note: Since Draw scans only your current directory, you must run Draw in the directory where your standard report files are stored.

When running Draw, you will be asked to select from a list of choices for each field. Since the list of choices is prepared from information in the standard report files, a default answer will be available for each question. You can accept a default value by pressing the RETURN key at each prompt.

As the interaction continues, each of your choices are verified. When all fields of a specification are obtained, the specification will be displayed on screen for verification. You can either save or discard the specification.

When the first specification is created and saved, you will be asked to enter the specification file name. The default is "draw.spec". If you choose to create additional specifications, the process will be repeated. All specifications created in the same session will be stored in the same specification file.

When you have completed all specifications, Draw will ask if your bar chart(s) should be generated. If you answer "no," Draw will exit from interactive mode, leaving you with just the specification file. If you answer "yes," you will be asked for the name of an output file; the default is "draw.out". Draw will generate a bar chart for each specification set in the specification file and exit.

5.4.2 Quick Interactive Mode

You may discover that the default values for the option statements, COMMENT, LEGEND, ORGANIZE, TITLE, XTITLE, YMIN, YMAX and YTITLE, are sufficient to execute Draw for your emulation. The -q option of the draw command specifies that these default values should be used. The resulting Draw execution can be completed more quickly than the normal interactive execution, since Draw will prompt you only for the required information.

Draw creates a specification file with the default name `draw.spec`. The output is placed in a file with the default name `draw.out`.

The following example illustrates using the `-q` option:

```
$ draw -q
Draw: EMPOWER V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95

Please respond after each => prompt.
Press RETURN to accept a default.
Type quit at any time to exit.

Please select one or more INPUT file names.
Available files in the current directory are:
  1) 16user.STD   2) 1user.STD   3) 2user.STD
  4) 4user.STD   5) 8user.STD
Enter * to select all the files.
=> 1
Please select an EVENT to be charted.
  1) Scenario      2) Function      3) Transaction
Default is 1) Scenario
=> 3
Please select one or more Transactions for the X axis.
Available X axis values are:
  1) Overall      2) Unspecified  3) query      4) update
Default is 1) Overall
=> 3 4
Please select one or more statistics for the Y axis.
Available Y axis values are:
  1) Total        2) Finish        3) Thruput    4) Median
  5) Average      6) Minimum        7) Maximum    8) Std-Dev
Default is 5) Average
=> 5 6 7
1 bar chart written to draw.out
Draw: normal completion.
$
```


during processing. Each BEGIN-END pair requires four statements that must appear on separate lines (see list below) and several optional statements.

Except in the case of input files, Draw does not distinguish between upper and lower case letters. The names of input files in the INPUT statement must be written exactly as they appear in your directory.

The required statements are:

- INPUT - input files (the standard reports)
- EVENT - event to be charted (scenario, function, or transaction)
- X - list of scenarios, functions, or transactions
- Y - performance measurements (average, maximum, P90, ...)

Each statement begins with a key word (statement name) and must be followed by a space and an equal sign:

Incorrect

Event= Transaction
Event=Transaction
Event Transaction
evENT = tranSAction

Correct

Event = Transaction
Event =Transaction
EVENT =TRANSACTION

Note: If you operate Draw interactively, Draw will ensure that your specification has no mistakes.

5.5.1 INPUT

INPUT specifies the standard report files that Draw will read. When one file is identified, up to 60 X-axis values can be specified. The Draw tool will generate a bar chart that compares different events in the same input file.

If multiple input files are specified, only one X-axis value is allowed. In this case, the bar chart compares an event across several emulations and up to 60 input files may be specified.

Examples:

```
INPUT = 16user.STD
INPUT = mix25.STD, mix50.STD
```

5.5.2 EVENT

EVENT specifies the event to be charted. The value of EVENT can be Scenario, Function, or Transaction and only one value can be defined for each specification.

Examples:

```
EVENT = Scenario
EVENT = Function
EVENT = Transaction
```

5.5.3 X

x specifies a list of events that will appear on the X-axis of the bar chart. The names in the x value must appear in the first column of all INPUT files under the selected heading in the EVENT statement. If multiple files are specified in the INPUT statement, the x value must contain only one item. If the INPUT statement identifies only one file, up to 60 items can be specified .

Examples:

```
X = query, update
X = Overall
```


[illegible]

Y = Thruput
Y = Minimum, Average, Maximum

TITLE = Minimum and Maximum Response Time

Journal Pre-proof

XTITLE = Number of Emulated Users

YMAX =10000000

LEGEND identifies each Y value with a single character and each bar will contain the corresponding LEGEND character. Performance measures have pre-defined default LEGEND characters.

Example:

LEGEND = * , ! , &

ORGANIZE describes the format of bars to be drawn on the chart. The value of ORGANIZE can be CLUSTERED or HIDDEN. In the CLUSTERED format, all bars corresponding to the same event are grouped together with different legend characters for each bar. With the HIDDEN format, all bars that correspond to the same event are combined into a single bar with different legend characters (the taller bars will hide behind the shorter bars).

ORGANIZE = HIDDEN

5.5.12 COMMENT

COMMENT = This chart created by DL

Table 1 Summary of the study design and participant characteristics

Monitoring script activity during load testing is very difficult without Monitor because you must activate the display option for each executing script or browse through script logs to discover problems. The display option requires a terminal or workstation for each displayed script which would be extremely expensive, and browsing through log files is very time-consuming and inefficient.

Benefits of using Monitor are:

- EMPOWER/CS V1.0.1

Copyright PERFORMIX, Inc. © 1995

For Bourne shell (sh) users, you can set your TERM variable by typing at the sh prompt:

```
TERM=vt100; export TERM
```

C-shell users would type at the csh prompt:

```
set term=vt100
```

or

```
setenv term vt100
```

Note: Of course, you would insert your appropriate terminal type in place of vt100.

Some curses implementations do not gracefully fail if you do not specify a valid terminal type before executing a curses application. Some of them simply fail to execute the application and return you to the UNIX shell with no indication that a problem occurred. Others may create more serious problems, such as dumping core. If a core dump occurs, you should verify that your terminal type is in the system's terminfo data base for System V or /etc/termcap for BSD UNIX. User-friendly curses implementations normally will print a message like the following if they detect that your terminal type is unknown or invalid:

```
<terminal type>: Unknown terminal type
```

or

```
I don't know enough about your <terminal type> terminal.
```

If Monitor does not recognize your terminal, the following message will appear:

```
I don't know enough about your terminal to run mon!
```


The `-e` option specifies that exited scripts will not be shown in Monitor views during the session. The default mode displays all scripts. The `-e` option only displays executing scripts which allows you to focus on scripts that are still running without sorting through "dead" scripts or deleting scripts from shared memory.

6.3.3 -k

The `-k` option allows you to kill all currently running scripts. After entering this option, a verification statement will appear. You can kill only those scripts belonging to you. See the following example:

\$ mon -k

This procedure kills processes owned by you that are attached to the EMPOWER shared memory segment. Continue?

If you answer `y` to the prompt while scripts belonging to another user are running, the following message will be displayed:

The following processes can't be killed because you're not the owner.

```
PID      Owner
1674     wendy
1681     wendy
2053     wendy
.....
```

...

If you want to force the kill of all scripts (even those not owned by you), you can log in as the super user and type "mon -k".

The `-o` option will start Monitor without displaying scripts that you do not own.

6.3.5 -r

The `-r` option allows you to remove the shared memory segment. This option will not start Monitor.

To remove the shared memory segment, you must own it (own either the first script to begin execution, or be the user who executes the `mon`, `xmon`, or `csmon` command, whichever occurs first), or be the super-user. If the shared memory segment is removed successfully, a message similar to the following will be displayed:

```
Shared memory segment ID 0 removed.
Shared memory semaphore ID 0 removed.
```

If scripts are running, you must use the `-k` option to kill all scripts before the shared memory segment can be removed.

If you are not the owner and you try to remove the shared memory segment with the `-r` option, the following message will be displayed:

The EMPOWER shared memory segment must be removed by its creator <creatorname>.

or

The EMPOWER/X shared memory segment must be removed by its creator <creatorname>.

You can force removal of the shared memory segment by logging in as the super-user and typing "mon -r".

There are 10 processes attached to the EMPOWER shared memory segment.

6.3.6 -w

6.3.7 -A

This function may not work on your terminal correctly (or at all) in the following three cases:

- EMPOWER/CS V1.0.1**


```
$ mon -S
```

EMPOWER V3.1.8 Serial#R00000-000 Copyright PERFORMIX, Inc. 1988-95
Maximum number of sessions available in this shared memory segment = 1024
Number of sessions currently attached to the shared memory segment = 1
Size of shared memory segment in bytes = <491440>
Owner of shared memory segment = <ownername>

The maximum number of sessions available is less than or equal to the maximum number of sessions that can be monitored. The default value is 1024.

The owner of the shared memory segment is the first user to execute the EMPOWER, EMPOWER/X, or EMPOWER/CS command that created the shared memory segment. This user will be either the first person to run a script or run Monitor.

\$ mon -v 3

\$ mon -i 2

\$ mon -s 64

\$ mon -s 2048

```
$ set path=( $path /usr/empower/bin )
```


Example view modes for each monitoring tool follow:

```

Mon Feb 11 10:43:16          EMPOWER MONITOR V3.1.8          (c) PERFORMIX, Inc. 1995
View 1 of 7  Script 1 of 8          Running          ScriptId Sorting ^  Interval 5

ScriptId  _Script State  _____LastXmit_____LastRcv
user1| manager| xmit| country.^[[kkA and w|[4hr country.^[[4l^H^[[A^H^H^H^[[An
user2| typist| xmit|^H^H^H^Hof their cou|hto come to the aid ^[[4l^M^[[B^[[K
user3| manager| think| o come to the aid ^M|hto come to the aid ^[[4l^M^[[B^[[K
user4| typist| rcv|^Mfor all good men^M|^[[K^[[4hfor all good men^[[4l^M
user5| typist| rcv|^Mfor all good men^M|[[4hfor all good men^[[4l^M^[[B^[[K
user6| manager| xmit|^Mvi^MiNow is the ti|^H~^[[B^H~^[[B^H~^[[B^H~^[[B^H~^[[H
user7| manager| rcv|ry.^[[kkA and women^[[6\344\240wo\355e\356^[[333\264l\210
user8| typist| xmit|.^[kkA and women^[:q|[A^H^H^H^[[An^[[4h and women^[[4l^H

```

```

Fri Apr 16 16:41:51          EMPOWER/X MONITOR V4.1.4          (c) PERFORMIX, Inc. 1995
View 1 of 10  Script 2 of 4          Running          ScriptId Sorting ^  Interval 5

ScriptId  __Script  ____State  C1  _____LastXmit  _____LastRcv
u1  script1  keystring  2  Mls^Mdate^Mwho^Mpwd^Mt  /Xstuff^J[empower@nomad]
u2  script1  textrcv  2  ^Mls^M  erylcode.Jeff^Jquerycode.S
u3  script1  disconnect  1  <LEFT_ALT>  P P P P P P P P P P P P P P P P
u4  script1  textrcv  2  ^Mls^Mdate^Mwho^M  6 10:18^J[empower@nomad]

```



```

Wed Jan  4 15:33:43      EMPOWER/CS MONITOR V1.0      (c) PERFORMIX, Inc. 1995
View 1 of 7  Script 1 of 4      Running      ScriptId Sorting ^  Interval 5

ScriptId  Script State  LastEvent  CurrentWindow
user1     foo type      <SysKeyPress>James Smith  Employee
user2     foo event      <SysKeyPress><LeftButtonPress>  Run
user3     foo event      <SysKeyPress>  Help
user4     foo type      <SysKeyPress>test.dat^M  Save As

```

You will begin your Monitor session in view mode. From this mode you can type '?' to display the help menu listing Monitor commands. The help menu is displayed in three help screens. You can press the space bar to toggle between the help screens or press a 'q' to exit the help screen. The three help screens for the EMPOWER Monitor are shown below.

Moving_____	Scrolling_____	Searching_____
h left	^u half screen up	/ search
j down	^d half screen down	\ reverse search
+ down	^f screen forward	n next search
k up	^b screen backward	filter out scripts
- up	^e one script down	Other_____
l right	^y one script up	^[forget input
nv to view n	Escaping_____	^l refresh
nG to script n	! shell	ni set interval to n
G to last script	T trace script	s change sort field
H highest on screen	E edit script	C screen capture
M middle of screen	V view log	d enable display
L last on screen	Z zoom to script	q quit
		? help

Press Spacebar to see next help, q to quit help...


```
Toggles_____
.  dots
a  anchor mode
b  bars
c  compressed
e  exit mode
o  owner mode
p  pause
r  relative time
w  wrap log before view
x  sort order

Press Spacebar to see first help, q to quit help...
```

EMPOWER/CS V1.0.1

6.6.1 Moving

j, +	Move the cursor n scripts down
k, -	Move the cursor n scripts up
h	Change the view to one view to the left wrapping at the end (If 1 is the current view, h will change the view to View 7.)
l	Change the view to one view to the right wrapping at the end (If 7 is the current view, l will change the view to View 1.)
n _v	Change the view to view n (If the current view is 2, 4 _v will change the view to view 4. The default value for n is current view + 1.)
nG	Move the cursor to the n th script scrolling as necessary (The default value for n is last.)
H	Move the cursor to the highest script on the current screen
M	Move the cursor to the middle script on the current screen
L	Move the cursor to the last script on the current screen

Not found below in <scriptid> column

No previous search pattern

The escape commands allow you to enter modes external to the view mode without quitting your Monitor session. After exiting an escape mode, you will return to view mode. The `shell` escape mode allows you to escape temporarily to a UNIX shell to execute shell commands; `trace` allows you to trace a script source code while the script executes; `edit` allows you to edit the script source code at the current execution location; `view` allows you to flush the script's log file buffer and view the log file; and `zoom`, only available for `mon`, displays the current screen of a script.

- ! shell escape mode command
- T trace escape mode command
- E edit escape mode command
- V view escape mode command
- Z zoom escape mode command (only available for mon)

The `zoom` escape mode works only if the following function is in your script source file:

```
Term(ZOOM, VT100 | LINES24 | AUTOWRAP)
```

You can enter the `shell` escape mode at any time. To enter other modes, you must move the cursor to the required script and press the escape command.

6.6.4.1 Trace Mode

When a script has been idle for an unusual amount of time, tracing its execution to see where the script is "blocked" can be very useful. The "t" command allows you to see what line in the script source file the script currently is executing.

Tracing a script assumes that you have used the `Beginsource()` and `Endsource()` functions properly. `Beginsource()` should be the first function executed whenever a script branches to a new source file. `Endsource()` should be executed just before a return. `Beginsource()` requires an argument - the name of the source file without the `.c` extension. `Endsource()` has no argument. Source files may be nested.

While tracing a script, you can use the "?" command to get the trace mode help menu:

```

    _Trace Help_

p  toggle pause
ni set interval to n
^l refresh
l  toggle line numbers
?  help
q  quit
Press q to quit help...

```


p	Pauses the cursor (Tracing will stop temporarily until the p command is pressed again.)
ni	Sets the update interval to n seconds
^1	Refreshes the trace screen
1	Displays the source script's line numbers on the trace screen (Pressing the 1 command again will turn the line number display off.)
?	Displays the help menu
q	Quits the Trace mode or quits the Help menu

The 'E' command allows you to invoke the "vi" editor on the script source file. The "vi" editor must be located either in /usr/bin/vi, or your UNIX \$PATH (\$path for csh users) must include the location of "vi".

When entering the edit mode, the cursor will be placed on the line currently executing. Once you have entered the edit mode, you are in the vi editor's command mode. You must know how to operate the "vi" editor before using the edit mode.

Note: Any changes made to the script will be included only after you recompile the script with the Csccl command and then execute it.

When a script has been idle for a long duration, looking at the script's log file is useful. The `v` command requests that the script flush its log file buffer before invoking the UNIX `view` command on the log file. Since the `v` command will cause

6.6.4.4 Zoom Mode

Before the zoom can occur, the user being "zoomed" must have been updating a copy of the screen initiated with the `Term()` function in a script. The first argument to `Term()` must be `ZOOM`, and the second must describe the terminal that the system under test (SUT) thinks it is updating, for example, `Term(ZOOM, VT100|LINES24|AUTOWRAP)`. The script will interpret the escape sequences sent to it and maintain a copy of the user's screen.

Frequency of the screen image is determined by the `zoom` interval, which is every five seconds by default. This interval can be changed by using the "i" command within the `zoom` mode.

EMPOWER/CS V1.0.1

p	Pauses the screen image (The screen update will stop temporarily until the p command is pressed again.)
ni	Sets the update interval to n seconds
^l	Refreshes the zoom screen
?	Displays the help menu
q	Quits zoom mode or quits the help menu

6.6.5.1 ˆ[

6.6.5.2 1

6.6.5.3 ni

`ni` sets the display interval to `n` seconds. The display will be repainted with current information every `n` seconds. The default interval value is five seconds, the maximum interval value is 3,600 seconds, and the minimum is one. If you specify an interval value exceeding the maximum, the interval will be set to the maximum

value. If you specify an interval value less than the minimum, the interval will be set to the default. When sorting by a field that may change often, we suggest that you do not set the time interval to a small value (such as 1i, 2i, or 3i).

6.6.5.4 s

s allows you to change the sort field (located on the second line of the header in Monitor) to a field on the current view. The default sort field is `ScriptId` where all the scripts will be listed by `ScriptId`. You can use the `h` and `l` keys to cycle through available fields. The `^[]` (Esc) key can be used to abort changing the sort field. Once you have selected the right field, press the `Enter` key to re-sort. Monitor pauses while the sort field is being selected.

When sorting on a field other than `ScriptId`, the `ScriptId` is used as a secondary sort field. Scripts will be sorted by the primary sort field first, and any scripts in which the primary sort field values are equal will be sorted again by the `ScriptId`. An exception to this process occurs when sorting by idle time; scripts that are in the "exit" state or the "suspend" state are sorted to the bottom since idle time is not applicable for these scripts. These scripts are sorted by `ScriptID` at the bottom of the list (i.e., after all scripts in other states).

To watch those scripts that are not progressing as anticipated, sorting by idle time is common. Scripts in a `RCV` state with considerable amounts of idle time may be encountering problems on the SUT that need to be fixed. If your script is in a think state listed under the `State` column, idle time is expected.

Monitor displays a "snapshot" of what occurs in executing scripts, but script data changes constantly. For this reason, scripts may appear out-of-sort if data changes between the times data are sorted and painted on the screen. Also, time values are displayed as HH:MM:SS.hh where hh indicates hundredths of a second. When sorting by time values, and the times grow so that hundredths of a second no longer display, scripts may seem out of sort because the time values appear the same but scriptids are not in order. Although the hundredths value is not displayed, it continues to be used for calculating the sort order.

Whenever many scripts appear to have completed execution, you may want to sort by `State` or `StatePattern` to help you isolate scripts that have terminated abnormally. Sorting by either of these fields while scripts execute is meaningless (and somewhat annoying) because data changes so often that it is obsolete as soon as it displays.

`c` allows you to capture the current screen into a specified file. You will be prompted with the file name the first time you issue the `c` command in the current session. The next time you invoke the `c` command, the current screen will append to the specified file.

d enables the display option for a script in the Monitor for EMPOWER. You can specify a port to watch the script execute on a separate terminal. The display option operates differently for the EMPOWER/X and EMPOWER/CS products. Refer to Sections X for details on operating the EMPOWER/X and EMPOWER/CS Monitor displays.

To initiate the display mode, move the cursor to a script in Monitor and enter the `d` command. Then, enter the name of a display TTY. You can specify a real TTY, for example `/dev/tty04`, or a pseudo TTY, for example `/dev/ttypl`, where `/dev` is optional. The base name of the TTY used for display mode can be found in Monitor View 4.

You should display to a terminal of the same type used by the emulated user. For instance, if the SUT is sending escape sequences to the script destined for a VT100, you should display to a terminal that can interpret these sequences.

The `Term()` function instructs EMPOWER to maintain a run-time screen image for use by any of the screen receive functions. This function also allows script execution to be viewed with the Zoom mode.

```
Term(ZOOM, VT100 | LINES24 | AUTOWRAP);
```

6.6.5.7 q

EMPOWER/CS V1.0.1

? displays the help menu.

When scripts complete execution, you can use the delete commands to remove them from shared memory which allows you to focus only on those scripts that are still running. To delete a script, you must either be the script owner or super-user and the script must be in the "exit" state.

Syntax for the delete commands is listed below.

- | | |
|-----|---|
| nDD | Delete n scripts starting with the current script (The default value for n is one.) |
| Ds | Delete all scripts on the current screen that belong to you |
| De | Delete all scripts in the "exit" state that belong to you |

If you do not own the script you attempt to delete, the following message will appear on the third line of the header:

Not owner!

If you specified an `nDD` command where `n` is too large, you will see the following message:

Not that many scripts

No exited scripts!

Not owner!

Script is already dead!

No live scripts!

By default, each script performs buffered writes to its log file to increase performance during multi-user tests. The flush commands can be used to signal a script to flush its log file buffer which is useful if you are browsing through logs and need to see the most recent information in the log. The V command requests that the script flush its log file automatically before invoking the UNIX view command.

nFF	Flush n running scripts starting with the current script (The default value for n is one.)
Fs	Flush all running scripts on the current screen that belong to you
Fa	Flush all running scripts that belong to you

Once a script is suspended, you can resume it using commands from *Monitor* or *Mix*. When resuming more than one script from *Monitor*, the scripts will resume in the background at intervals specified by the `nRt` command. When resuming from

nRR	Resume n suspended scripts (The default value for n is one.)
Rs	Resume all suspended scripts in the current screen that belong to you
Ra	Resume all suspended scripts that belong to you
nRt	Set the resume interval to n seconds (The default value for n is five.)

You may suspend a script while it executes if you are the owner of that script. Once a script is suspended, it will remain suspended until it is resumed.

- | | |
|-----|--|
| nSS | Suspend n scripts (The default value for n is one.) |
| Ss | Suspend all scripts on current screen that belong to you |
| Sa | Suspend all scripts that belong to you |

Your script may become blocked in a receive state during your EMPOWER Monitor session. If so, you can interrupt the script with the interrupt commands. Interrupting will force a timeout in the script, and the script will continue if possible. If you have set a large timeout value, interrupting is useful if you do not want to wait for the timeout interval to expire before executing subsequent script transactions.

The syntax for the interrupt commands is listed below.

- | | |
|-----|--|
| nII | Interrupt n blocked scripts starting with the current script (The default value for n is one.) |
| Is | Interrupt all blocked scripts on the current screen that belong to you |
| Ia | Interrupt all blocked scripts that belong to you |
| nIt | Set the interrupt interval (The default value for n is zero.) |

If you interrupt a script that is not blocked, you will receive the following message:

Script is not in a rcv state!

6.7.7 Joining

The Join feature of Monitor (which applies only to the Monitor for EMPOWER) allows you to become the emulated user of a blocked script. You simply select a script and your display will become that of the emulated user. You can interact with the application on the SUT in an attempt to revive it, determine what went wrong, or simply put the application back on track so the script can continue. With the Join feature, you will not have to abort tests simply because one or two scripts fail, and developers who need to resolve problems that occur only under load can debug them while a test continues. All keystrokes entered during the join are recorded so you can incorporate them for subsequent script runs.

A script can be joined in three ways: Two ways are from Monitor and are performed dynamically; one way is from within the script and occurs at a predetermined location. These methods are discussed in more detail in the following sections. Before a join from Monitor can occur, a script must be blocked in a receive state or be suspended. You can use the `ss` command from `mon` to force a script to suspend if it does not reach a blocked state voluntarily.

Note: You should join a script only from a terminal that is the same type set by the emulated user on the SUT. Output from the SUT and the function keys that you enter will work only when the terminal types are consistent.

You can join a script in a predetermined location by including the `Join()` function in a script. When your script encounters the `Join()` function, a joined session will begin. Such a script must be executed in the foreground, preferably with the `-d` or `-D` option, because `Join()` will be ignored if the script runs in the background.

The keystrokes and responses that occurred during a joined session will be built into a portion of the corresponding script. The script's prefix will be "mon." followed by the script ID with a .c extension. EMPOWER will attempt to group the keystrokes into logical transactions. If the .c file exists, the new portion of the script will be appended to the file. You can use the .x or .c files built during the joined session to aid in patching a script so that it can handle new behavior from an application on the SUT.

The toggle commands affect overall operation of Monitor.

- . converts all two-character control characters to dots making the screen easier to read and allowing more data to fit on the screen.
- a anchors the script. The < sign is used to indicate an anchored script and is displayed in the column between the ScriptId and the Script name. When sorting on fields that often change, the anchor is useful for continuing to monitor a particular script.
- b adds a vertical line between fields to discriminate more easily between columns of data. If the screen will be repainted often, do not use this command because printing the column separator will slow painting the screen.
- c compresses all views of one script onto the screen. To choose a script to compress, move the cursor to that script and press 'c'. For mon and csmon, press 'c' again to toggle back to the original, uncompressed view of all scripts. For xmon, press 'c' a second time to remove blank lines. If your screen originally was not large enough to display information from all 10 xmon views, this process will allow you to view more of the current script's status

- e toggles exit mode. The default mode displays all scripts. The alternate mode displays only executing scripts. This command allows you to focus on scripts that are still running without having to sort through "dead" scripts or delete scripts from shared memory.
- o toggles owner mode. The default mode displays all user scripts. The alternate mode displays only scripts that belong to you.
- p pauses display, since script information changes often. You may move to another script when paused, but you cannot scroll the screen. The ? (help) command also will pause Monitor.
- r toggles between absolute and relative times. This is useful when displaying timestamps in view 5 of mon and csmon and view 6 of xmon.
- w toggles between wrapping and not wrapping logs before viewing. Wrapping is useful if your log includes responses from the SUT which are too long to be viewed by a system editor or which contain unprintable characters.
- x toggles between ascending (^) and descending (v) sort order.

Seven different views are available for `mon`, ten are available for `xmon`, and seven for `csmon`. Each view displays a screen with a table of information for all scripts currently running or recently exited and all views display information for the same set of scripts. Each script requires one line on your screen. If you have more scripts than lines on your screen, you can scroll to the remaining scripts.

When Monitor starts, it enters view mode in a "Loading" state until it has initialized to begin monitoring. Next, it will go into a "Waiting" state until it has scripts to monitor. As scripts begin to execute, it will go into a "Running" state. Other possible states are "Paused" and "Sorting". These states are displayed in view mode.

The seven views available in the Monitor for EMPOWER (mon) are described in this section.

View 1 of `mon` displays the state of each script and the data transmitted and received by each script.

ScriptId	A unique identifier specified in the script table and used by the Mix tool (If you run a script from the command line, the ScriptId will be the same as the script name.)
Script	The name of the compiled script (The source version of the script typically is found in script.c.)
State	The current state of the script's execution (xmit, rcv, suspend, think, and exit)
LastXmit	The last characters transmitted to the SUT by the script (Characters scroll from right to left as they are transmitted so that the rightmost character is the last character transmitted.)
LastRcv	The last characters received from the SUT by the script (Characters scroll from right to left as they are received so that the rightmost character is the last character received.)

EMPOWER/CS V1.0.1

Sorting by State is useful when most scripts are in the same state or entering the same state, as when scripts are exiting or suspending.

When many control characters are displayed, entering '.' (dot) mode will allow more characters to be displayed in the `LastXmit` and `LastRcv` fields.

6.7.1.2 EMPOWER Monitor View 2

View 2 of mon displays detailed information of script execution states, idle time, and the number of warning and timeout messages for each script. This view often is used for debugging scripts.

Mon Feb 11 10:47:37			EMPOWER MONITOR V3.1.8		(c) PERFORMIX, Inc. 1995		
View 2 of 7 Script 1 of 8			Running		Idle Sorting v Interval 5		
ScriptId	Script	State	StatePattern	Idle	NWarn	NTo	ToCondition
user5	typist	rcv	XXX	13.97			15 continue
user7	manager	rcv]	2.27		1	15 continue
user1	manager	think	1.83	1.33		1	15 continue
user8	typist	xmit	who^M	.58			15 continue
user6	manager	xmit	for all good men^M	.56			15 continue
user3	manager	xmit	^H^Hof their country.^[.54		1	15 continue
user2	typist	think	1.78	.51		1	15 continue
user4	manager	think	1.33		1	4	15 continue

StatePattern	Additional information relative to the script execution state, such as the string being transmitted (Xmit), the string being looked for (Rcv), the current think value (Think), or the exit status (exit)
Idle	The time, in seconds, that the script has been idle or the time since the shared memory segment was last updated by the script
NWarn	The number of EMPOWER warnings issued to the script (early pattern matches, etc)
NTo	The number of EMPOWER timeouts issued to the script (i.e. when an expected rcv string never arrives)
ToCondition	The number of seconds before a timeout will occur and the action to be taken upon timeout

View 3 of `mon` displays script execution completion percentage, the script source file name, the log file name, the communication port used by the script, and any script notes.

Line	The line in the script source (.c) file currently executing
NLine	The number of lines in the source (.c) file
Pct	The percentage of script completion
Source	The name of the script source file (may be a function source file)
Log	The log file to which the script is writing
Port	The port on the RTE through which the script is communicating
Note	A user-defined note placed in the script by using the Note() function (A note can be placed at any location in the script and changed as appropriate.)

View 4 of `mon` displays general information about the scripts, such as the owner of the script, think time distribution, type rate, and the amount of I/O traffic information. I/O information can be used in workload analysis.

```

Fri Apr 16 14:05:20          EMPOWER MONITOR V3.1.8          (c) PERFORMIX, Inc. 1995
View 4 of 7  Script 1 of 4          Running          ScriptId Sorting ^  Interval 5

ScriptId  __Script  __Pid  ____Owner  NCXmit  ____NCRcv  _IO  _Type  _Disp  _____ThinkTime
user1 example1  499  empower      68      5441  80      ttyq2  uniform 10. 25.
user2      ss    501  empower      67      4464  66      uniform 10. 25.
user3      db    503  empower      66      2708  41      uniform 10. 25.
user4      comp  506  empower      68      2765  40      uniform 10. 25.

```

Pid	The process ID of the script
Owner	The log in ID of the script's owner, (i.e., the person who started the script)
NCXmit	The total number of characters transmitted to the SUT
NCRcv	The total number of characters received from the SUT
IO	I/O ratio of the script or the number of characters received for every one character transmitted
Type	The typing speed of the script
Disp	The display terminal, if applicable
ThinkTime	The think time distribution and parameters

6.9.1.5 EMPOWER Monitor View 5

View 5 of mon displays times of various events as well as elapsed time information.

Users often can estimate load test completion time by looking at this view.

Mon Feb 11 10:54:06		EMPOWER MONITOR V3.1.8		(c) PERFORMIX, Inc. 1995				
View 5 of 7 Script 4 of 8		Running		ScriptId Sorting ^ Interval 1				
ScriptId	__Script	__Start	_Suspend	__Resume	_BeginSc	__EndSc	____Exit	_Elapsed
user1	manager	10:45:33			10:45:33	10:51:10	10:51:10	2:54.39
user2	typist	10:45:48			10:45:48	10:51:26	10:51:26	2:38.50
user3	manager	10:46:03			10:46:03	10:51:41	10:51:42	2:23.12
user4	manager	10:46:18	10:53:38	10:53:46				18.44
user5	typist	10:46:33			10:46:33	10:52:11	10:52:11	1:53.86
user6	manager	10:46:48			10:46:48	10:52:26	10:52:26	1:38.63
user7	manager	10:47:03			10:47:03			7:02.11
user8	typist	10:47:18			10:47:18	10:52:56	10:52:56	1:08.41

Start	The time script execution began
Suspend	The time of the most recent Suspend() in the script
Resume	The time the script resumed execution after a Suspend()
BeginSc	The time of Beginscenario() execution, usually before any transactions are executed
EndSc	The time of Endscenario() execution
Exit	The time of script completion
Elapsed	The time elapsed since the most recent script activity (If a script is exited, it will indicate the time since exiting. If a script has begun a scenario, as shown above, it will indicate the time elapsed since the scenario was started.)

View 6 of `mon` displays function response times as functions complete. With this view, users often can determine how the SUT is handling the workload.

```

Mon Feb 11 10:51:11      EMPOWER MONITOR V3.1.8      (c) PERFORMIX, Inc. 1995
View 6 of 7  Script 1 of 8      Running      CurrFnRt Sorting v  Interval 1

ScriptId  __Script  ____NFn  _AveFnRt  _____LastFn  LastFnRt  _____CurrFn  CurrFnRt
user7  manager                2  1:27.23          vi  1:26.45          vi  2:43.59
user2  typist                 2  1:27.08          vi  1:26.14          vi  1:26.09
user3  manager                2  1:26.98          vi  1:25.94          vi  1:11.20
user5  typist                 2  1:27.06          vi  1:26.01          vi   40.00
user6  manager                2  1:26.90          vi  3:11.00          vi   25.22
user8  typist                 1  3:11.00          vi  1:28.19
user4  manager                3  1:27.47          vi
user1  manager                3  1:27.47          vi

```

- | | |
|----------|---|
| NFn | The number of functions completed (Each function is a set of transactions.) |
| AveFnRt | Average response time of functions completed |
| LastFn | The last function completed (LastFn and CurrFn will be displayed only when you specify a BeginFunction() and EndFunction() pair.) |
| LastFnRt | The response time of the last function completed |
| CurrFn | The function currently executing |
| CurrFnRt | The response time of the function currently executing |

6.9.1.7 EMPOWER Monitor View 7

View 7 of mon displays transaction response times as transactions complete.

Mon Feb 11 10:52:10		EMPOWER MONITOR V3.1.8		(c) PERFORMIX, Inc. 1995			
View 7 of 7 Script 1 of 8		Running		CurrXr Sorting v Interval 5			
ScriptId	Script	NXr	AveXrRt	LastXr	LastXrRt	CurrXr	CurrXrRt
user7	manager	1	.13	vi	.13	who	
user8	typist	28	.46	vi	.13	vi	
user6	manager	30	.48	vi	.09	vi	.56
user4	manager					date	
user5	typist	33	.49	ls	.92		
user3	manager	33	.51	ls	.92		
user2	typist	33	.50	ls	.92		
user1	manager	33	.51	ls	.95		

NXr	The number of transactions Xmit()-Rcv() pairs completed
AveXrRt	The average response time of transactions completed
LastXr	The last transaction completed (LastXr and CurrXr will be displayed only when you specify a BeginTransaction() and EndTransaction() pair.)
LastXrRt	The response time of the last transaction completed
CurrXr	The transaction currently executing
CurrXrRt	The response time of the transaction currently executing

The ten views available in the Monitor for EMPOWER/X (*xmon*) are described in this section.

View 1 of `xmon` displays the state of each script and the data transmitted and received by each script.

ScriptId	A unique identifier specified in the Mix script table (If you run a script from the command line, the ScriptId will be the same as the script name.)
Script	The name of the compiled script (The source version of the script is typically found in script.c.)
State	The current state of the script's execution (keystring, textrcv, suspend, think, or exit)
Cl	The active client (application) on the SUT
LastXmit	The last characters transmitted to the SUT by the script (Characters scroll from right to left as they are transmitted so that the rightmost character is the last character transmitted.)
LastRcv	The last characters received from the SUT by the script (Characters scroll from right to left as they are received so that the rightmost character is the last character received.)

Sorting by `State` is useful when most scripts are in the same state or entering the same state, as when scripts are exiting or suspending.

EMPOWER/CS V1.0.1

View 2 of `xmon` displays detailed information of script execution states, idle time, and the number of warning and timeout messages for each script. This view often is used for debugging scripts.

```

Fri Apr 16 16:41:23      EMPOWER/X MONITOR V4.1.4      (c) PERFORMIX, Inc. 1995
View 2 of 10  Script 1 of 4      Running      ScriptId Sorting ^  Interval 5

ScriptId  __Script  _____State  Cl  _____StatePattern  __Idle  NTo  Limit  __ToCondition
u1  script1  internal  2  QueryColors  1.67  10  10  continue
u2  script1  internal  1  GrabPointer  .53  10  10  continue
u3  script1  textrcv  2  ConfigureWindow  .87  10  10  continue
u4  script1  internal  2  QueryColors  1.67  10  10  continue

```

StatePattern	Additional information relative to the script execution state, such as the string being transmitted (<code>keystring</code>), the string being looked for (<code>rcv</code>), the current think value (<code>think</code>), or the exit status (<code>exit</code>)
Idle	The time, in seconds, that the script has been idle, or the time since the script last updated the shared memory segment
NTo	The number of EMPOWER/X timeouts issued to the script (i.e. when an expected <code>rcv</code> string never arrives)
Limit	The limit of the number of EMPOWER/X warnings and timeouts issued before script termination
ToCondition	The number of seconds before a timeout will occur and the action to be taken upon timeout


```

Fri Apr 16 16:41:29          EMPOWER/X MONITOR V4.1.4    (c) PERFORMIX, Inc. 1995
View 3 of 10 Script 2 of 4          Running          ScriptId Sorting ^ Interval 5

ScriptId __Script __Client Cl NCl __Seq _____LastSMsg _____LastCMsg
    u1  script1      2    2    45  ConfigureNotify      ConfigureWindow
    u2  script1      2    2    73      FocusIn      TranslateCoordinates
    u3  script1      2    2   165      NoExpose      ImageText8
    u4  script1      2    2    57      Reply      GetGeometry

```

Copyright PERFORMIX, Inc. © 1995

View 6 of `xmon` displays times of various events as well as elapsed time. Users often can estimate load test completion time by looking at this view.

```

Fri Apr 16 16:41:37      EMPOWER/X MONITOR V4.1.4      (c) PERFORMIX, Inc. 1995
View 6 of 10  Script 2 of 4      Running      ScriptId Sorting ^  Interval 5

ScriptId  __Script  __Start  __Suspend  __Resume  __BeginSc  __EndSc  ____Exit  __Elapsed
u1  script1  16:40:54                .90                41.90
u2  script1  16:40:54                .33                41.94
u3  script1  16:40:56                .51                39.64
u4  script1  16:40:58                .30                38.19

```

Start	The time script execution began
Suspend	The time of the most recent <code>Suspend()</code> in the script
Resume	The time the script resumed execution after a <code>Suspend()</code>
BeginSc	The time of <code>Beginscenario()</code> execution, usually before any transactions are executed
EndSc	The time of <code>Endscenario()</code> execution
Exit	The time of script completion
Elapsed	The time elapsed since the most recent script activity (If a script exited, it will indicate the time since exiting. If a script has begun a scenario, it will indicate the time elapsed since the scenario was started.)

View 7 of `xmon` displays function response times as functions complete. With this view, users often can determine how the SUT is handling the workload.

NFn	The number of functions completed (Each function is a set of transactions.)
AveFnRt	Average response time of functions completed
LastFn	The last function completed (LastFn and CurrFn will be displayed only when you specify a BeginFunction() and EndFunction() pair.)
LastFnRt	The response time of the last function completed
CurrFn	The function currently executing
CurrFnRt	The response time of the function currently executing

6.9.2.8 EMPOWER/X Monitor View 8

View 8 of `xmon` displays transaction response times as transactions complete.

```

Fri Apr 16 16:41:41      EMPOWER/X MONITOR V4.1.4      (c) PERFORMIX, Inc. 1995
View 8 of 10      Script 2 of 4      Running      ScriptId Sorting ^      Interval 5

ScriptId  __Script  ____NXr  _AveXrRt  _____LastXr  LastXrRt  _____CurrXr  CurrXrRt
u1  script1      1      .32      return      .32      date      2.44
u2  script1      1      .41      return      .41
u3  script1      4      1.75      pwd      .88      tty      7.52
u4  script1      2      1.55      date      2.69      who      2.09

```

NXr	The number of transactions KeyString() -TextRcv() pairs completed (e.g. data base queries, UNIX shell commands)
AveXrRt	The average response time of transactions completed
LastXr	The last transaction completed (LastXr and CurrXr will be displayed only when you specify a BeginTransaction() and EndTransaction() pair.)
LastXrRt	The response time of the last transaction completed
CurrXr	The transaction currently executing
CurrXrRt	The response time of the transaction currently executing

Note: NXr, AveXrRt, LastXrRt, and CurrXrRt will have values only when a BeginTransaction()-EndTransaction() pair has been specified.

View 9 of `xmon` displays statistics that indicate the number of events and other messages sent by the X clients and X servers.

NR	The number of Request (R) messages sent by the X client(s)
Nr	The number of reply (r) messages sent by the X server
Ne	The number of event (e) messages sent by the X server
No	The number of error (o) messages sent by the X server
NCMsgs	The number of messages sent by the X client(s)
NSMsgs	The number of messages sent by the X server(s)

View 10 of `xmon` displays the number of bytes of messages sent by the X clients and the X servers.

```

Fri Apr 16 16:41:48          EMPOWER/X MONITOR V4.1.4      (c) PERFORMIX, Inc.
1995
View 10 of 10  Script 2 of 4          Running          ScriptId Sorting ^  Interval 5

ScriptId __Script __NRBytes __NrBytes __NeBytes __NoBytes ____NCBytes ____NSBytes
u1  script1      23928      11148      4224          0          23928      15372
u2  script1      23232      11148      4000          0          23232      15148
u3  script1      25476      11148      4864          0          25476      16012
u4  script1      24608      11148      4416          0          24608      15564

```

- | | |
|---------|---|
| NRBytes | The number of bytes of the Request (R) messages sent by the X client(s) |
| NrBytes | The number of bytes of the reply (r) messages sent by the X server |
| NeBytes | The number of bytes of the event (e) messages sent by the X server |
| NoBytes | The number of bytes of the error (o) messages sent by the X server |
| NCBytes | The number of bytes of the messages sent by the X client(s) |
| NSBytes | The number of bytes of the messages sent by the X server(s) |

6.9.3 EMPOWER/CS Monitor Views

The seven views available for the Monitor for EMPOWER/CS (`csmon`) are described in this section.

6.9.3.1 EMPOWER/CS Monitor View 1

View 1 of `csmon` displays the state of each script and the data transmitted and received by each script.

Wed Jan 4 15:33:43		EMPOWER/CS MONITOR V1.0.1		(c) PERFORMIX, Inc. 1995	
View 1 of 7		Script 1 of 4		Running	
				ScriptId Sorting ^	
				Interval 5	
ScriptId	Script	State	LastEvent	CurrentWindow	
user1	foo	type	<SysKeyPress>James Smith	Employee	
user2	foo	event	<SysKeyPress><LeftButtonPress>	Run	
user3	foo	event	<SysKeyPress>	Help	
user4	foo	type	<SysKeyPress>test.dat^M	Save As	

ScriptId	A unique identifier specified in the script table and used by the Mix tool (If you run a script from the command line, the ScriptId will be the same as the script name.)
Script	The name of the compiled script (The source version of the script typically is found in a <code>.c</code> file)
State	The current state of the script's execution
LastEvent	The last user interactions transmitted to the database server by the script (Characters scroll from right to left as they are transmitted so that the rightmost character is the last character transmitted.)
CurrentWindow	The last <code>CurrentWindow()</code> function.

Monitor enforces that `ScriptIds` are unique by allowing a script to "steal" the Monitor slot of another script having the same `ScriptId`. If you run one load test

after the first is completed, the second run will "re-use" the first Monitor slots. Users must be careful to designate different script IDs.

Sorting by State is useful when most scripts are in the same state or entering the same state, as when scripts are exiting or suspending.

StatePattern	Additional information relative to the script execution state
Idle	The time, in seconds, that the script has been idle or the time since the shared memory segment was last updated by the script
NDBerror	The number of database errors issued to the script
NTto	The number of EMPOWER timeouts issued to the script (i.e., when an expected WindowRcv() never arrives)
NExec	The number of Exec() statements in the script that were executed,
NRFetch	The number of fetches from the database after the Exec()

6.9.3.3 EMPOWER/CS Monitor View 3

View 3 of `csmon` displays script execution completion percentage, the script source file name, the log file name, the data type used by the script, and amount of think time.

```

Wed Sep 28 14:09:40      EMPOWER/CS MONITOR V1.0.1      (c) PERFORMIX, Inc. 1995
View 3 of 7  Script 1 of 4      Running      ScriptId Sorting ^  Interval 5

ScriptId _Script Line NLine _Pct _____Source _____Log Type_____Thinktime
foo      foo      33      63 52.4      foo      foo 5.0 uniform 1.0 2.5

```

Line	The line in the script source (.c) file currently executing
NLine	The number of lines in the source (.c) file
Pct	The percentage of script completion
Source	The name of the script source file (may be a function source file)
Log	The log file to which the script is writing
Type	The typing speed of the script
Thinktime	The think time distribution and parameters

6.9.3.4 EMPOWER/CS Monitor View 4

View 4 of `csmon` displays general information about the scripts.

```

Wed Sep 28 14:09:40      EMPOWER/CS MONITOR V1.0.1  (c) PERFORMIX, Inc. 1995
View 4 of 7  Script 1 of 4      Running      ScriptId Sorting ^  Interval 5

ScriptId __Script __Pid __Display____Arguments __ToCondition ____Note
foo      foo      704      ergy              300 continue

```

Pid	The process ID of the script
Display	The display terminal, if applicable
Arguments	The number of arguments to the scripts
ToCondition	The number of seconds before a timeout will occur and the action to be taken upon timeout
Note	A user-defined note placed in the script by using the Note() function (A note can be placed at any location in the script and changed as appropriate.)

6.9.3.5 EMPOWER/CS Monitor View 5

View 5 of `csmon` displays times of various events as well as elapsed time information. Users often can estimate load test completion time by looking at this view.

Wed Sep 28 14:09:40		EMPOWER/CS MONITOR V1.0.1		(c) PERFORMIX, Inc. 1995				
View 5 of 7 Script 4 of 8		Running		ScriptId Sorting ^ Interval 1				
ScriptId	__Script	__Start	_Suspend	__Resume	_BeginSc	__EndSc	__Exit	_Elapsed
user1	manager	10:45:33			10:45:33	10:51:10	10:51:10	2:54.39
user2	typist	10:45:48			10:45:48	10:51:26	10:51:26	2:38.50
user3	manager	10:46:03			10:46:03	10:51:41	10:51:42	2:23.12
user4	manager	10:46:18	10:53:38	10:53:46				18.44
user5	typist	10:46:33			10:46:33	10:52:11	10:52:11	1:53.86
user6	manager	10:46:48			10:46:48	10:52:26	10:52:26	1:38.63
user7	manager	10:47:03			10:47:03	-		7:02.11
user8	typist	10:47:18			10:47:18	10:52:56	10:52:56	1:08.41

Start	The time script execution began
Suspend	The time of the most recent <code>Suspend()</code> in the script
Resume	The time the script resumed execution after a <code>Suspend()</code>
BeginSc	The time of <code>Beginscenario()</code> execution, usually before any transactions are executed
EndSc	The time of <code>Endscenario()</code> execution
Exit	The time of script completion
Elapsed	The time elapsed since the most recent script activity (If a script has exited, the time since exiting will be indicated. If a script has begun a scenario, as shown above, the time elapsed since the scenario was started will be indicated.)

View 6 of `csmón` displays function response times as functions complete. With this view, users often can determine how the SUT is handling the workload.

```

Wed Sep 28 14:09:40      EMPOWER/CS MONITOR V1.0.1      (c) PERFORMIX, Inc. 1995
View 6 of 7  Script 1 of 8      Running      CurrFnRt Sorting v  Interval 1

ScriptId  _Script  ____NFn  _AveFnRt  _____LastFn  LastFnRt  _____CurrFn  CurrFnRt
user7  manager                2  1:27.23          vi  1:26.45          vi  2:43.59
user2  typist                2  1:27.08          vi  1:26.14          vi  1:26.09
user3  manager                2  1:26.98          vi  1:25.94          vi  1:11.20
user5  typist                2  1:27.06          vi  1:26.01          vi   40.00
user6  manager                2  1:26.90          vi  3:11.00          vi   25.22
user8  typist                1  3:11.00          vi  1:28.19
user4  manager                3  1:27.47          vi
user1  manager

```

- | | |
|----------|---|
| NFn | The number of functions completed (Each function is a set of transactions.) |
| AveFnRt | Average response time of functions completed |
| LastFn | The last function completed (LastFn and CurrFn will be displayed only when you specify a BeginFunction() and EndFunction() pair.) |
| LastFnRt | The response time of the last function completed |
| CurrFn | The function currently executing |
| CurrFnRt | The response time of the function currently executing |

6.9.3.7 EMPOWER/CS Monitor View 7

View 7 of `csmon` displays transaction response times as transactions complete.

```

Wed Sep 28 14:09:40      EMPOWER/CS MONITOR V1.0.1      (c) PERFORMIX, Inc. 1995
View 7 of 7  Script 1 of 8      Running      CurrXr Sorting v  Interval 5

ScriptId  _Script  _____NXr  _AveXrRt  _____LastXr  LastXrRt  _____CurrXr  CurrXrRt
user7  manager      1      .13      vi      .13      who
user8  typist      28      .46      vi      .13      vi
user6  manager      30      .48      vi      .09      vi      .56
user4  manager      date
user5  typist      33      .49      ls      .92
user3  manager      33      .51      ls      .92
user2  typist      33      .50      ls      .92
user1  manager      33      .51      ls      .95

```

NXr	The number of transactions completed
AveXrRt	The average response time of transactions completed
LastXr	The last transaction completed
LastXrRt	The response time of the last transaction completed
CurrXr	The transaction currently executing
CurrXrRt	The response time of the transaction currently executing

[This page intentionally left blank]

09697994.102600


```
$ EMPOWER/bin/gvinstall -s
```

7.2 Architecture

EMPOWER/GV consists of three parts:

- *Global Variables* which are created and stored in the shared memory of the UNIX script driver
- *Global Variable Commands* which are executed from the shell prompt on the UNIX script driver to access variables
- *Global Variable Functions* which are placed in the script .c file to allow a script to access a variable during script execution

Global variables are created by a Global Variable command (`gv_init`) and stored in the shared memory of the UNIX script driver. This shared memory is accessible by both Global Variable commands and functions. A variable remains in shared memory until it is removed and it will retain its value after script execution. If the UNIX script driver is rebooted, all global variables are removed. Since global variables are stored in shared memory, your system must have the System V Shared Memory Facility installed to use the Global Variables tool.

You can enter Global Variable commands at the shell prompt of the UNIX script driver terminal. These commands also may be used during script execution with the Mix tool. Global Variable functions are placed in a script .c file during editing and can be used only when the script is compiled with Scc, Xscc, or Csc. (*Note:* For EMPOWER or EMPOWER/X users, scripts executed with Preview or Xpreview will not recognize the Global Variable functions.)

Global Variable commands and functions are used to create, allocate, read, update, test, and protect global variables. Changes to the value of a variable are made immediately. You can update a variable by assigning a new value or by applying a mathematical operation to the current value of the variable. You may test the value of a variable and use it in conjunction with the C language `while` statement so that an operation continues while the variable has a certain value. You also may protect a variable with a Global Variable command or function so that it can be accessed only by the script that protected it.

7.3 Global Variable Commands

Global Variable commands are UNIX shell commands used from the shell prompt on the UNIX script driver and from shell scripts to control global variables. They control access to a variable, read the current value of a variable, specify a new value for a variable, test the value of a variable, and control the shared memory segment used by global variables. All Global Variable commands begin with "gv_."

When a Global Variable command accesses a variable, the operation completes before any other command or function may access the variable. (*Note:* This uninterrupted operation is referred to as "atomic" referencing.)

When a Global Variable command (except for `gv_test`) executes successfully, the return code is set to zero. If a command results in an error, an error message is printed to the standard error output destination, or `stderr`, which is usually the UNIX script driver. With the exception of the `gv_test` command, an error will set the return code to one.

If you enter a command with an improper number of arguments, a usage message displays the correct syntax of the command.

Syntax help for each command can be obtained by entering the command name followed by a hyphen:

```
$ gv_add -  
Usage:  
    gv_add name value  
$
```


Valid test relations used for relational comparisons in Global Variable commands are listed below:

Equality Relations

<u>Relation String</u>	<u>Relation Test</u>
"=="	variable equal to value
"!="	variable NOT equal to value
"<"	variable less than value
"<="	variable less than or equal to value
">"	variable greater than value
">="	variable greater than or equal to value

Bit-Wise Relations

<u>Relation String</u>	<u>Relation Test</u>
" & "	variable AND value
" ! & "	variable NOT AND value
" "	variable OR value
" ! "	variable NOT OR value
" ^ "	variable EXCLUSIVE OR value
" ! ^ "	variable NOT EXCLUSIVE OR value

Logical Relations

<u>Relation String</u>	<u>Relation Test</u>
" ? "	variable is non-zero (non-NULL for str)
" ! "	variable is zero (NULL for str)

Many variable types are supported by the Global Variable Library. The name of the type is used as a parameter in the `gv_init` command. EMPOWER/GV includes support for type name aliases which allows a shorter type name to be used.

<u>Type Name</u>	<u>Alias</u>	<u>Description</u>
char	[none]	character
unsigned char	uchar	unsigned character
int	[none]	integer
unsigned int	uint, unsigned	unsigned integer
short int	short	short integer
unsigned short	intushort, ushort int, unsigned short	unsigned short integer
long int	long	long integer
unsigned long int	ulong, ulong int, unsigned long	unsigned long integer
float	[none]	single precision floating point
double	long float	double precision floating point
string	str	character string
parallel	par	parallel global variable

Variable type "parallel" refers to parallel Global Variables. Parallel global variables allow a specified number of scripts in a multi-user emulation to execute a series of transactions at the same time. All other scripts in the emulation will block while the first scripts are working in parallel. The blocked scripts wait until the parallel scripts have completed execution.

EMPOWER/CS V1.0.1

my_init (Creates the variable with the specified name type and initial value)

11. The following table shows the number of people who visited the National Museum of Natural History in Washington, D.C., from 1990 to 2000. The number of visitors is in millions.

The `gv_read` command returns the current value of the specified variable to the standard output destination (which is usually the terminal). In the following example, 3 is the current value of `users`.

```
$ gv_read users
3
```

The `gv_getparallel` command returns the current value of the specified parallel variable.

Example:

```
$ gv_getparallel workers
5
```

The `gv_stat` command sends to the standard output destination a table showing the name, type, and value of the specified variable. The table also shows the number of scripts that have allocated the variable and identifies the variable's protector. For example:

```
$ gv_stat users
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Name          Type          Value          Allocated Protector
-----
users          int              3              0      NONE
```

If `gv_stat` is executed without an argument, information will be listed for all variables:

```
$ gv_stat
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Name          Type          Value          Allocated Protector
-----
cust          int              100            0  CONSOLE
para          parallel         6              0
foo           int              1              1
```

7.3.5 Updating a Variable

The Global Variable commands for updating a variable are listed below:

<code>gv_write</code>	Assigns a new value to the variable
<code>gv_setparallel</code>	Assigns a new value to the parallel variable
<code>gv_inc</code>	Increases the value of the variable by one
<code>gv_dec</code>	Decreases the value of the variable by one
<code>gv_unparallel</code>	Increases the value of the parallel variable by one
<code>gv_add</code>	Adds the specified amount to the current value of the variable
<code>gv_sub</code>	Subtracts the specified amount from the current value of the variable
<code>gv_mul</code>	Multiplies the current value of the variable by the specified amount
<code>gv_div</code>	Divides the current value of the variable by the specified amount
<code>gv_mod</code>	Performs a modulo operation on the variable
<code>gv_lshift</code>	Performs a bit-wise shift to the left on the variable
<code>gv_rshift</code>	Performs a bit-wise shift to the right on the variable
<code>gv_and</code>	Applies bit-wise AND masking to the variable
<code>gv_or</code>	Applies bit-wise OR masking to the variable
<code>gv_xor</code>	Applies bit-wise EXCLUSIVE-OR masking to the variable

Each of these commands returns the current value of the variable to the standard output destination before performing the specified operation. For example,


```
$ gv_write users 6
5
$ gv_sub users 4
6
$ gv_read users
2
$
```

The Global Variable commands for testing a variable are `gv_test`, `gv_waitwhile`, `gv_waituntil`, and `gv_parallel`. These commands compare the current value of a variable to a specified value or condition.

The `gv_test` command simply tests the variable as specified. Results of the `gv_test` command are listed below:

If Comparison	Print to Standard Output	Set the Return Code to
is true	1	0
is false	0	1
fails due to error	error message	2

Example:

```
$ gv_read users
2
$ gv_test users "==" 2
1
$ echo $status
0
$ gv_inc users
2
$ gv_read users
3
$ gv_test users "==" 2
0
$ echo $status
1
$
```

Note: \$? replaces \$status when displaying the return value for the Bourne shell and the Korn shell.

The `gv_waitwhile` command waits while the test condition is true. The `gv_waituntil` command waits until the test condition is true. These commands do not send a message to the standard output destination. If the command is successful, the return code is set to zero. If the variable cannot be referenced, it is set to one. The difference between `gv_waitwhile` and `gv_waituntil` is illustrated below.

`gv_waitwhile("a", "<", 3)`

results in:

```
a = 0 -> wait
a = 1 -> wait
a = 2 -> wait
a = 3 -> do not wait
a = 4 -> do not wait
a = 5 -> do not wait
```

`gv_waituntil("a", "<", 3)`

results in:

```
a = 0 -> do not wait
a = 1 -> do not wait
a = 2 -> do not wait
a = 3 -> wait
a = 4 -> wait
a = 5 -> wait
```


These commands are described below:

<code>Gv_alloc()</code>	Allocates access to a variable
<code>Gv_free()</code>	De-allocates access to a variable
<code>Gv_protect()</code>	Prevents other scripts from accessing a variable until the <code>Gv_unprotect()</code> function is executed
<code>Gv_unprotect()</code>	Removes protection from a variable, allowing other scripts to access the variable

If a script references a global variable, the `Gv_alloc()` function should be the first function in the script. The `Gv_alloc()` function has two parameters: variable name and variable type. The variable type must match the type specified when the variable was created with the `gv_init` command. For example:

```
Gv_alloc("users", "int");
```

You can de-allocate a global variable with `Gv_free()`. If a global variable is not de-allocated with `Gv_free()`, it automatically will de-allocate when the script exits.

If a script tries to de-allocate a variable that has not been allocated to the script, an error will occur. The following error messages occur for any function when the specified variable has not been allocated to the script or when the variable does not exist:

```
Global variable not currently allocated
Global variable does not exist
```

The parameter of the `Gv_protect()` and `Gv_unprotect()` functions is the name of the variable (e.g., `users` from the example above). Only the script that protects a variable may unprotect it.

7.4.2 Reading a Variable

The Global Variable functions for reading a variable are `Gv_read()`, `Gv_readv()`, `Gv_getparallel()`, and `Gv_stat()`. The `Gv_read()` and `Gv_readv()` functions return the current value of a variable. The `Gv_getparallel()` function returns the current value of the specified parallel variable. The `Gv_stat()` function supplies status information about variables.

The `Gv_read()` function returns the current value of the specified global variable. Generally, this value is stored in a new variable. For example:

```
int curcount;
Gv_alloc("count", "int");
curcount = Gv_read("count");
```

The value of `count` is read and stored in an integer variable called `curcount`.

The value returned by the `Gv_read()` function is always an integer. If you need to use the return value and the actual value of a global variable is not an integer, you must use the function name ending with a "v" to obtain the correct value. In this case, an additional parameter is used to indicate where to store the value.

For example, if you want a script to read the current value of a parameter called `balance` but the value is not an integer, use the `Gv_readv()` function as in the following script file excerpt:

```
float curbalance;
Gv_alloc("balance", "float");
Gv_readv("balance", &curbalance);
```

In this example, the `Gv_readv()` function retrieves the current value of the variable `balance` and stores it in the floating point variable `curbalance`.

<code>Gv_write()</code> , <code>Gv_writev()</code>	Assigns a new value to the variable
<code>Gv_setparallel()</code>	Assigns a new value to the parallel variable
<code>Gv_inc()</code> , <code>Gv_incv()</code>	Increases the value of the variable by one
<code>Gv_dec()</code> , <code>Gv_decv()</code>	Decreases the value of the variable by one
<code>Gv_unparallel()</code>	Increases the value of the parallel variable by one
<code>Gv_add()</code> , <code>Gv_addv()</code>	Adds the specified amount to the current value of the variable
<code>Gv_sub()</code> , <code>Gv_subv()</code>	Subtracts the specified amount from the current value of the variable
<code>Gv_mul()</code> , <code>Gv_mulv()</code>	Multiplies the current value of the variable by the specified amount
<code>Gv_div()</code> , <code>Gv_divv()</code>	Divides the current value of the variable by the specified amount
<code>Gv_mod()</code> , <code>Gv_modv()</code>	Performs a modulo operation on the variable
<code>Gv_lshift()</code> , <code>Gv_lshiftv()</code>	Performs a bit-wise shift to the left on the variable
<code>Gv_rshift()</code> , <code>Gv_rshiftv()</code>	Performs a bit-wise shift to the right on the variable
<code>Gv_and()</code> , <code>Gv_andv()</code>	Applies bit-wise AND masking to the variable
<code>Gv_or()</code> , <code>Gv_orv()</code>	Applies bit-wise OR masking to the variable
<code>Gv_xor()</code> , <code>Gv_xorv()</code>	Applies bit-wise EXCLUSIVE-OR masking to the variable

When an update function is executed, the original value of the variable is saved if the variable is an integer, or copied to a pointer location if the variable is not an integer. After the operation is performed and the resulting value is assigned, the original value of the variable is returned as an integer.

```
Gv_write("count", 12);
```

```
float curbalance;
Gv_alloc("balance", "float");
Gv_writev("balance", 120.75, &curbalance);
```

```
Gv_inc("count");
```

```
float curbalance;
Gv_alloc("balance", "float");
Gv_incv("balance", &curbalance);
```

EMPOWER/CS V1.0.1


```
int curcount;
Gv_alloc("count", "int");
curcount = Gv_mul("count", 4);
```

```
Gv_alloc("count", "int");
Gv_inc("count");
Gv_waitwhile("count", "<", 20);
```


7.5 Using Global Variables Commands and Functions

Global variables are accessed by using Global Variable commands and functions with the variable name as an argument. The format for Global Variable commands is `gv_name`, and for functions, `Gv_name()`. Both must be used to make effective use of global variables.

```
$ gv_init customer int 100
```

To use a variable within a script, you should insert the `Gv_alloc()` function in the beginning of the script .c file to access the variable, for example:

The `Gv_alloc()` function permits the script to access and control the variable `customer` in other global variable functions, for example:

EMPOWER/CS V1.0.1


```
!gv_init users int 0
use table
start s1 s2 s3
wait
quit
```

7.5.2 Global Variables Used By Multiple Scripts

However, simultaneous access to a set of variables may cause resource contention problems in your UNIX kernel when multiple scripts try to access or protect the same variable. Problems generally arise from careless use of the `Gv_protect()`, `Gv_unprotect()`, `Gv_waitwhile()`, and `Gv_waituntil()` functions.

EMPOWER/CS V1.0.1

A circular protection of variables may result, causing two scripts to pause indefinitely, as in the following situations:

- ☐ script one protects variable one
- ☐ script two protects variable two
- ☐ script one tries to protect variable two
- ☐ script two tries to protect variable one

You always must be aware of the effects that variable protection will have on other running scripts. When a variable is to be used by multiple scripts but is protected by one script, you should design your scripts so that the variable becomes unprotected at some point. Other scripts attempting to read or write to the variable will block until the variable is unprotected.

When several scripts employ the `Gv_waitwhile()` or `Gv_waituntil()` function for the same test condition, some scripts may pause indefinitely. This situation may occur because the test condition becomes true (or false) for such a short time that some scripts may never see the condition for which they are waiting.

For example, a script may include the following:

```
/* Increment the global variable "users" by one */
Gv_inc("users");
/* Wait while "users" is less than 3 */
Gv_waitwhile("users", "<", 3);
/* Decrement the global variable "users" by one */
Gv_dec("users");
```

When the value of `users` becomes three, this script will return from the `Gv_waitwhile()` function. The value of `users` then will be decreased by one so that its new value is two. If other scripts are using the same `Gv_waitwhile()` function, they may not see the value of `users` reach three since it is immediately decreased by this script. Therefore, the other scripts may be blocked indefinitely. This error is avoided most easily by designing test conditions that are not true (for `Gv_waituntil()`) or false (for `Gv_waitwhile()`) for only a short time.

7.5.3 Setting Type Rate and Think Time With Global Variables

Global Variable commands and functions can be used to set emulated type rate and think time range from the UNIX script driver prior to running a test. Global variables are created to specify type rate and minimum and maximum think times. Functions in the scripts then use these variables to set type rate and think time range.

For example, to set the type rate at two characters per second and the minimum and maximum values of the think time distribution at one and five seconds, use the following Global Variable commands:

```
$ gv_init typerate int 2
$ gv_init thinkmin int 1
$ gv_init thinkmax int 5
```

Then, place the following statements in the beginning of the script .c file:

```
Gv_alloc("typerate", "int");
Gv_alloc("thinkmin", "int");
Gv_alloc("thinkmax", "int");

Typerate(Gv_read("typerate"));
Thinkuniform(Gv_read("thinkmin"), Gv_read("thinkmax"));
```

Each time the script is executed, a new type rate and think time distribution may be specified. If the `Typerate()` and `Thinkuniform()` functions are placed within a loop in the script, new values may be set as the script executes.

7.5.4 Examples

The following sections present introductory examples for using global variables. For simplicity, the example scripts shown in the following sections are EMPOWER scripts. Global variables also may be used in EMPOWER/X and EMPOWER/CS scripts.


```

$ gv_init synch int 0
$ gv_protect synch
$ gv_stat
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Name                Type                Value                Allocated Protector
-----
synch                int                0                    0
CONSOLE
$ s1 &
[1] 3058
s1 ready
$ s2 &
[2] 3060
$ s3 &
[3] 3062
s2 ready
s3 ready
$ gv_stat
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Name                Type                Value                Allocated Protector
-----
synch                int                0                    3
CONSOLE
$ gv_unprotect synch
$ gv_stat
EMPOWER/GV V3.1.8, Serial#R00000-000, Copyright PERFORMIX, Inc. 1988-95
Name                Type                Value                Allocated Protector
-----
synch                int                3                    3 NONE

```


7.5.4.3 Unique Numbers

In this example, a global variable provides unique numbers for a set of scripts that delete records from a database. Since the global variable maintains its value after the scripts have executed, the scripts may be run multiple times without resetting the database.

This example uses the `gv_init` command and the `Gv_alloc()` and `Gv_inc()` functions. The `gv_init` command creates the desired global variable and sets the initial value. The `Gv_alloc()` function permits the script to use the variable and ensures that the variable type specified in the script matches the type specified in the `gv_init` command. The `Gv_inc()` function reads the current value of the variable and increments it by one.

First, the global variable `customer` is created at the command line on the UNIX script driver. The variable is an integer and the initial value is 100:

```
$ gv_init customer int 100
```

The following script source code deletes five records from the database:

```
/* dbdelete.c */
Empower(argc, argv)
int argc;
char *argv[];
{
    int i;
    int thiscustomer;
    char buf[10];
    Thinkuniform(2,5);      /* think 2 to 5 seconds */

    /* Allocate the global variable "customer" to this script
       (if "customer" does not exist yet, exit) */
    Gv_alloc("customer", "int");
    Beginscenario("dbdelete");
    Rcv("$ ");
    Xmit("dbstartup^M");
    Rcv("> ");
}
```

(continued on following page...)


```

/* delete five records from the database */
for (i = 0; i < 5; i++) {
    /* get a customer number to use and increment
       for the next script */
    thiscustomer = Gv_inc("customer");
    sprintf(buf, "%d", thiscustomer);

    Mxmit("delete ", buf, "^M", "");
    Rcv("> ");
}
Xmit("quit^M");
Rcv("$ ");
Xmit("exit^M");
Rcv("connection closed.^J");
Endscenario("dbdelete");
}

```

Two scripts containing this source code are executed. The first script is run with the display option (-d) which is shown below:

```
$ dbdelete -d rlogin:sut s1 &
$ dbdelete rlogin:sut s2 &
$

Welcome to SUT

$ dbstartup
Speedy Database started at 12:56:01 EST 10/07/90
> delete 100
record for customer 100 deleted
> delete 102
record for customer 102 deleted
> delete 103
record for customer 103 deleted
> delete 105
record for customer 105 deleted
> delete 107
record for customer 107 deleted
> quit
$ exit
connection closed.
$
```


Before the emulation is executed, the variable must be created:

The value of `filetoken` is never used because we are protecting (not testing or incrementing) the variable. Therefore, its initial value is unimportant. Also, the variable does not need to be re-initialized for subsequent runs of the emulation.

```

/* wordproc_1.c */
/* Allocate the global variable "filetoken" to this script
   (if "filetoken" does not exist yet, exit) */
Gv_alloc("filetoken", "int");
Beginscenario("wordproc_1");
Rcv("$ ");
/* do some transactions - not detailed here */
Xmit("cd documents^M");
.
.
.
Rcv("$ ");
/* Protect variable "filetoken" from other scripts
   (Only one script can protect filetoken at a time) */
Gv_protect("filetoken");
/* do the exclusive edit - not detailed here */
Xmit("vi proposal^M");

```

EMPOWER/CS V1.0.1


```
.  
.  
Xmit(":wq^M");  
Rcv("$ ");  
  
/* Unprotect variable "filetoken"  
   (This allows another script to protect "filetoken") */  
Gv_unprotect("filetoken");  
Xmit("exit^M");  
Rcv("connection closed.^J");  
Endscenario("wordproc_1");
```

7.5.4.5 Indefinite Benchmark Duration

In this example, several scripts are designed to run indefinitely so that they may be terminated from the UNIX script driver. The `kill` command of `Mix` does not ensure that scripts will terminate gracefully because the script simply disconnects from the system under test (SUT). You can assign a global variable that instructs scripts to complete the current series of transactions and then quit the SUT application before disconnecting.

To set up this type of script control, each script should be edited to check the designated global variable before each series of transactions begins. This verification is accomplished by enclosing each series of transactions in a `while` loop and using the `Gv_test()` function which tests the value of the global variable. If the comparison is true, then the script will proceed with the transactions.

First, the global variable is created:

```
$ gv_init workflag int 1
```

The variable is given an initial value of 1. The name `workflag` was chosen because the variable specifies whether or not to continue working.

The following interaction produces these results, using the `gv_test` command at the shell prompt:

```
$ gv_init a int 5
$ gv_test a "<" 1
0
$ gv_test a ">" 1
1
```

The script code using `Gv_test()` is shown below:

```
Gv_alloc("workflag", "int");
Beginscenario("worker");
Rcv("$ ");
/* enter the SUT application */
Xmit("officemail^M");
Rcv("om> ");
while ( Gv_test("workflag", "==", 1) ) {

    /* enter a series of transactions - not detailed here */
    Xmit("....");

    .
    .
    Rcv("om> ");
}

/* quit the SUT application */
Xmit("quit^M");
Rcv("$ ");
/* exit from the SUT */
Xmit("exit^M");
Rcv("connection closed.^J");
Endscenario("worker");
```



```
$ gv_write workflag 0
1
```

Before running the scripts again, `workflag` will have to be reset. You are not required to specify `workflag` as an integer because it already exists:

```
$ gv_init workflag 1
```

Use the `gv_init` command to create a new variable and assign a value to the variable. Unless the specified variable currently is allocated by a script, `gv_init` also can assign a new value to an existing variable.

Therefore, `gv_init` generally is used between tests, and `gv_write` is used during a test.

In the following example, a set of scripts will execute a series of transactions a specified number of times. As in the previous example, the scripts are set up so that the value of a global variable is checked before the transactions execute.

A global variable function is created to count down how many times the series of transactions executes. The function `Gv_dec()` is used to decrement the count each time the transactions execute.

First, the variable is created and given the initial value 600, which is the total number of times the series of transactions will be executed during the multi-user emulation.

```
$ gv_init work int 600
```

Each script first gains access to the variable `work` with the `Gv_alloc()` function. Then, the `Gv_dec()` function decrements the value of `work` by one. To determine if the new value of `work` is greater than zero, a test is performed with the `while` statement. If the value of the variable is greater than zero, the contents of the loop (the series of transactions) are executed.

Each of these scripts will contain the following:

```
Gv_alloc("work", "int");
Beginscenario("worker");
Rcv("$ ");
/* enter the SUT application */
Xmit("officemail^M");
Rcv("om> ");
while (Gv_dec("work") > 0) { /* there's work to do */
    /* enter a series of transactions - not detailed here */
    Xmit("....");
    .
    .
    Rcv("om> ");
}
/* quit the SUT application */
Xmit("quit^M");
Rcv("$ ");
/* exit from the SUT */
Xmit("exit^M");
Rcv("connection closed.^J");
Endscenario("worker");
```


Once the value of `work` becomes zero (i.e., the series of transactions has been executed 600 times), each script will terminate as it gets to the `while` loop. As each script falls through the `while` loop, the value of `work` is decremented to an increasingly large *negative* number. Since the `while` loop is set up to execute only if `work` is greater than zero, this negative number creates no problems.

To run the multi-user test again, the variable `work` must be reset. If desired, you may initialize `work` to a different value, and you are not required to specify `work` as an integer since it already exists:

```
$ gv_init work 120
```

7.5.4.7 Script Synchronization

In this example, a set of scripts is synchronized before the scripts execute commands on the SUT. A global variable is created and incremented as each script starts. The function `Gv_waitwhile()` pauses each script until all scripts have started.

The `Gv_waitwhile()` function makes the script pause as long as the specified relational comparison is true. When the comparison becomes false, script execution continues past the `Gv_waitwhile()` statement.

The global variable that stores the number of executed scripts is initialized:

```
$ gv_init users int 0
```

Each script first gains access to the variable `users` with the `Gv_alloc()` function. Then, the `Gv_inc()` function increments the value of `users` by one and `Gv_waitwhile()` tests the value of `users`.

Each script includes the following:

```
/* Allocate the global variable "users" to this script
   (if "users" does not exist yet, exit) */
Gv_alloc("users", "int");

Beginscenario("datetest");

Rcv("$ ");

/* Increment the global variable "users" */
Gv_inc("users");

/* Wait while "users" is less than 3 */
Gv_waitwhile("users", "<", 3);

/* do some transactions - not detailed here */
Xmit("cd documents^M");
.
.
.
Rcv("$ ");

Xmit("exit^M");
Rcv("connection closed.^J");

Endscenario("datetest");
```

To run the multi-user emulation again, the variable `users` must be reset since it will retain a value of three after the first execution. You are not required to specify that `users` is an integer since it already exists.

Example:

```
$ gv_init users 0
```


7.5.4.8 Limited Script Activity in Multi-User Emulation

In this example, parallel Global Variables control execution of multiple scripts. Parallel variable commands and functions allow a subset of emulated users to execute a portion of the script at a given time. Specifically, during an emulation involving 500 users, a portion of the test may be executed in parallel by only 50 users. Each user will execute one script. The Global Variable `worker` is created from the UNIX script driver shell as a parallel type variable with an initial value of 50:

```
$ gv_init worker parallel 50
```

The following script is executed by all users (some detail is left out for brevity):

```
login()

{/* login transactions - not detailed here */}

Empower(argc, argv)
    int argc;
    char *argv[];
{
    Gv_alloc("worker", "parallel");
    login();
    Beginscenario("example");

    /* ... numerous transactions */

    Gv_parallel("worker");

    /* ... limited portion of transactions */

    Gv_unparallel("worker");

    /* ... numerous transactions */

    Endscenario("example");
}
```


As the 50th user executes the `Gv_parallel()` function, the value of the variable `worker` is decremented to zero. As each of the first 50 scripts reaches the `Gv_unparallel()` function, the value of `worker` is incremented to be greater than zero, so that one of the waiting scripts can execute the limited portion of transactions. This process ensures that only the first 50 scripts will execute the specified transactions at the same time.

Creating scripts to read input files on the UNIX script driver is a common practice. Such files contain names, addresses, phone numbers, etc., used for emulating database queries and updates. For such an emulation, each user would need unique data, so, each script must read from a different file. A test of 100 users would need 100 input files. Considerable effort often is required to create the 100 files, especially if you must run tests with different numbers of users.

`Fioshare()` in a script presumes execution of the `fioshare` command at the UNIX script driver's shell prompt. The `fioshare` command creates a global variable that contains the offset for the next byte to be read from a shared file. The value of the global variable (offset) remains between tests, so you can continue to read an input file from the point left by the previous test. Saving the offset in this manner

is useful in tests that corrupt a database on the SUT. The ability to avoid the same transactions means you do not have to restore the database before every test.

You must execute the `fioshare` command if you want to resume reading from the beginning of the input file. For this reason, `fioshare` often is run from Mix command files that set up for a new test.

For example, assume we require scripts to read commands from the following input file `cmds`:

```
date
who
ls /bin
ps -ef
grep xxx /etc/passwd
```

The following command will create the global offset for the file:

```
$ fioshare cmds
```

A segment in the script described above might look like the following example. Each execution of this script will read different lines.

```
Fioshare("cmds");
Fioreadline("cmds");
while (FIOLEN != -1) {
    Mxmit(FIOBUFFER, "^M", "");
    Rcv("$ ");
    Fioreadline("cmds");
}
```

The global offset is stored in a global variable and the global variable's name is the inode of the shared file. This condition can be confirmed by typing the `ls -i` command with an argument of the shared file and then running the `gv_stat` command.

For example:

[illegible]

To discontinue sharing a file, the `Fiounshare()` function and `fiounshare` shell command are used. Neither the function nor the command remove the global variable offset for the file. They simply mark the global variable as being unusable for further File I/O functions.

The `Fiounshare()` function disables the sharing of the file offset only for the script that executes the function. The `fiounshare` shell command disables sharing of the file offset for all scripts currently sharing the file.

To remove the global variable offset, you must use the `gv_rm` shell command to remove the global variable named in the `gv_stat` output listed above. For example:

\$ gv_rm -f 59449

7.5.4.10 Script Scenario Selection

For this example, we use a global variable to select types of scenarios to execute. Multiple copies of a single script are used and the script contains all of the desired scenarios.

The test includes a mix of following scenarios:

<u>Scenario</u>	<u>Count</u>
receptionist	1 user
database	3 users
manager	1 user
visitor	Remainder of users

A single script with functions containing each set of transactions will implement this mix of scenarios. By using a global variable called `users`, the `EMPOWER()` function will select a unique ID for the script. Global Variables allow this test to execute without passing arguments to the script through the Mix table.

The `Gv_inc()` function increments the value of `users`. Before it is incremented, the value of `users` is saved in the local variable `myid`. Each script has a unique value of `myid` which is used in a `switch()` statement to select the proper scenario to run.

First, the global variable `users` is initialized with a zero value. *Note:* This initialization must be performed prior to each run:

```
$ gv_init users int 0
```

Several copies of the script are started with Mix. The first script to call `Gv_inc()` returns a value of 0 to `myid`, so it calls the `receptionist()` function. The next three (`myid` values 1, 2, and 3) call the `database()` function. The next script (`myid` = 4) calls the `manager()` function. The remainder of scripts (`myid` = 5, 6, etc.) call the `visitor()` function.

In the following script, the actual transactions for the four scenarios are left out for brevity:


```
login()
{
Rcv("login: ");

Xmit("userid^M");
Rcv("word:");

Xmit("passwd^M");
Rcv("$ ");
}

manager()
{ /* manager transactions go here */ }

receptionist()
{ /* receptionist transactions go here */ }

database()
{ /* database transactions go here */ }

visitor()
{ /* visitor transactions go here */ }

Empower()
{
    int myid;

    Gv_alloc("users", "int");
    myid = Gv_inc("users");

    Timeout(30, CONTINUE);
    Thinkuniform(1,3);

    login();

    switch(myid) {
        case 0: Beginscenario("receptionist");
               receptionist();
               Endscenario("receptionist");
               break;
```

(continued on following page...)

:

7.5.4.11 Scripts Controlled By One Script

In the following example, 11 identical scripts execute simultaneously and are controlled by the first executed script. This controlling script will not interact with SUT applications so its communication port to the SUT can be specified as "pseudo:sh -i" on the command line. The controlling script sets the global variables that are accessed by the other scripts.

First, four global variables are initialized:

```
$ gv_init control int 0
$ gv_init quitflag int 0
$ gv_init users int 0
$ gv_init duration int 3600
```

The first script to call `Gv_inc()` becomes the controller. It pauses (with `Gv_waitwhile()`) while the other ten scripts start. As each script starts, this controlling script increments the variable `users` and waits while `users` is less than

ten. Once these ten scripts have started, the controller script uses `Gv_read()` to retrieve the value of global variable `duration`. The controller script then executes a sleep delay for the number of seconds specified by the `duration` variable.

All other scripts execute the function `do_invoice`. This function includes a `while` loop which checks the value of global variable `quitflag`. If `quitflag` has the value zero, the function continues to execute.

When the controller script concludes the sleep delay, it increments `quitflag`, causing the other scripts to return from `do_invoice`. The controller script uses `Gv_waitwhile()` to pause until the other scripts have completed. Each of the other scripts decrements the variable `users`. When the value of `users` is zero, the controller script returns from `Gv_waitwhile()`, re-initializes control and `quitflag`, and exits.

Each script would include the following:

```
login()
/* login transactions - not detailed here */

do_invoice()
{
    /* do some transactions - not detailed here */
    Xmit("...");
    .
    .
    Rcv("$ ");
}

Empower()
{
    Gv_alloc("control", "int");
    Gv_alloc("quitflag", "int");
    Gv_alloc("users", "int");
```

(continued on following page...)


```

if (Gv_inc("control") == 0) {
Gv_waitwhile("users", "<", 10);
Gv_alloc("duration", "int");
Sleep(Gv_read("duration"));
Gv_inc("quitflag");
Gv_waituntil("users", "==", 0);
Gv_write("control", 0);
Gv_write("users", 0);
Gv_write("quitflag", 0);
}
else {
login();

Beginscenario("invoice");

Gv_inc("users");

while(Gv_test("quitflag", "==", 0))/* check for quit indicator */
do_invoice();

Gv_dec("users");

Endscenario("invoice");

Xmit("exit^M");
Rcv("connection closed.^J");
}
}

```

7.5.4.12 Scripts With Collective Throughput

In the following example, scripts use global variables to control throughput of transactions the scripts submit to the SUT. Script execution is controlled at the shell by setting the value of the global variable `workflag`.

First, global variables `workflag` and `transactions` are initialized:

```
$ gv_init workflag int 0
$ gv_init transactions int 0
```


When each script returns from `Gv_waitwhile()`, it uses the `Time()` function to record system time in the local variable `starttime`. Then, the script checks `workflag` with `Gv_read()` and if the value of `workflag` is one, enters a `while` loop to begin a transaction. The transaction executes, the variable `transactions` is incremented, and the time is stored in the variable `curtime` with `Time()`.

Script execution terminates when the shell command `gv_write` resets the value of `workflag` to zero.

An example script follows:

```
#define THRUPUT 10.0    /* transactions per second */
login()
{/* login transactions - not detailed here */}

Empower()
{
    struct timevalue starttime, curtime;
    Gv_alloc("transactions", "int");
    Gv_alloc("workflag", "int");

    Timeout(30, CONTINUE);
    Thinkuniform(1,3);
    login();

    Gv_waitwhile("workflag", "==", 0);
    Beginscenario("worker");

    Time(&starttime);
    while(Gv_test("workflag", "==", 1)) {
        /* do a transaction - not detailed here */
        Xmit("...");
        Rcv("$ ");

        Gv_inc("transactions");

        Time(&curtime);
        if ((Gv_read("transactions") /
            Difftime(&starttime, &curtime)) > THRUPUT)
            Sleep(Range(0,3));
    }
    Endscenario("worker");

    Xmit("exit^M");
    Rcv("");
}
```


In the following example, two scripts process transactions with a database application. After the first script has completed database operations, a control number received from the SUT is saved in a global variable for the second script's use. The first script copies the control number from the response buffer `RBUFFER` into the global variable `mailbox`. The second script copies the saved control number from the mailbox into a transmit buffer.

```
$ gv_init mailbox str ""
```

```
char control[GVSTRINGMAX + 1];
Gv_alloc("mailbox", "str");

Rcv("$ ");

/* do a transaction - not detailed here */
Xmit("...");
/* align the control number at the beginning of RBUFFER */
Scan("Control number is: ");

/* read control number response into RBUFFER */
Rcv("$ ");

strncpy(control, RBUFFER, 10);

/* put control number in global variable for script2 */
Gv_writev("mailbox", control, (char*)NULL);
```



```
Gv_alloc("mailbox", "str");
char control[GVSTRINGMAX + 1];

Rcv("$ ");

/* read the control number string left in the global variable */
Gv_readv("mailbox", control);

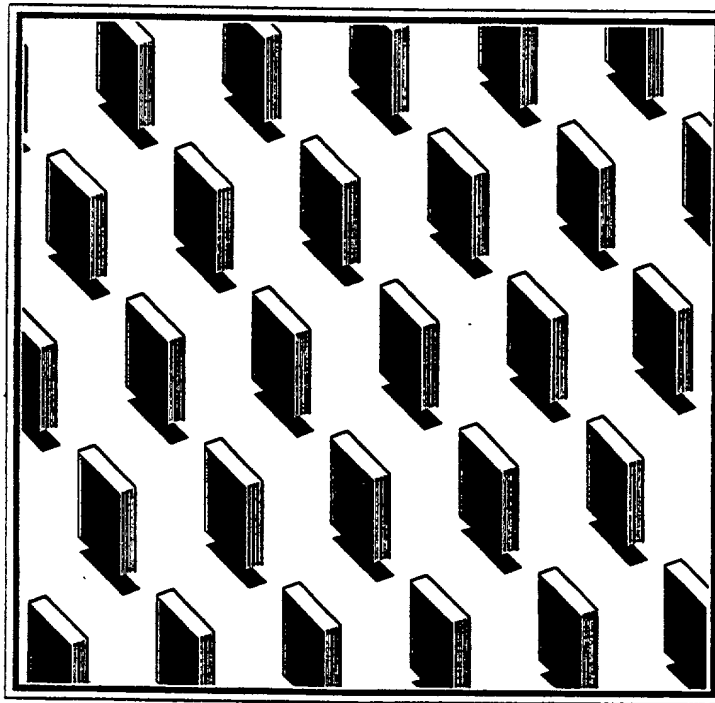
/* do a transaction with the new string */
Mxmit(control, "^M", "");
Rcv("$ ");
```


0907

1. Introduction

EMPOWER

Reference



f o r E M P O W E R / C S



 **PERFORMIX**

03637944-102600

LIMITATION OF LIABILITY

PERFORMIX makes no warranty or representation of any kind, either expressed or implied, with respect to this software, updates, or documentation, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. This software, updates, and documentation are provided 'AS IS'. The entire risk as to their quality, performance, and results is assumed by you. Some states do not allow the exclusion of implied warranties, so the above limitation may not apply to you.

In no event will PERFORMIX be liable to you for any damages whatsoever (including but not limited to direct, indirect, special, incidental, or consequential damages) arising out of the use or inability to use the software, updates, or documentation even if PERFORMIX has been advised of the possibility of such damages. In particular, PERFORMIX is not responsible for any damages including but not limited to those resulting from lost profits or revenue, loss of use to the computer program, loss of data, claims by third parties, or for other similar damages. Some states do not allow the execution or limitation of liability for consequential or incidental damages, so the above limitation may not apply to you.

COPYRIGHT

The PERFORMIX documentation and the software are copyrighted and protected by both the Universal Copyright Convention and the Berne Convention. All rights are reserved. No part of this documentation nor the software may be copied, reproduced, translated, or transmitted in any form or by any means except as expressly described in your license agreement.

U.S. GOVERNMENT RESTRICTED RIGHTS

If this software and documentation are acquired by or on behalf of a unit or agency of the United States Government this provision applies. This software and documentation: (a) were developed at private expense, and no part of it was developed with government funds, (b) are a trade secret of PERFORMIX, Inc. for all purposes of the Freedom of Information Act, (c) are "commercial computer software" and "computer software documentation" subject to limited utilization as provided in the contract between the vendor and the government entity, and (d) in all respects are proprietary data belonging solely to PERFORMIX, Inc.

The enclosed software and documentation are provided with RESTRICTED AND LIMITED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR 52.227-14 (June 1987) Alternate III(g)(3)(June 1987), FAR 52.227-19(June 1987), or DFARS 252.227-7013 (c)(1)(ii)(October 1988), as applicable. Contractor/Manufacturer is PERFORMIX, Inc., 8200 Greensboro Drive, Suite 1475, McLean, VA 22102. Unpublished-rights reserved under the copyright laws of the United States.

EMPOWER/CS is a trademark of PERFORMIX, Inc. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Windows is a trademark of Microsoft Corporation. IBM is a registered trademark of International Business Machines Corporation. UNIX is a registered trademark of UNIX Systems Laboratories, Inc. Oracle is a registered trademark of Oracle Corporation. Sybase is a registered trademark of Sybase, Inc. All other trademarks are the property of their respective holders.

Software Version 1.0.1, User's Guide Version 1.0.1



PERFORMIX, Inc.

8200 Greensboro Drive, Suite 1475

McLean, VA 22102

(703) 448-6606 (phone)

(703) 893-1939 (fax)

1.0	Introduction	1-1
1.1	Organization of the EMPOWER/CS User's Guides	1-1
1.2	Organization of this Manual	1-2
1.3	User's Guide Conventions.....	1-2
<hr/>		
2.0	Reference	2-1
<hr/>		
3.0	Error Messages	3-1
<hr/>		
4.0	EMPOWER/CS Technical Support.....	4-1

This manual contains technical reference material for the EMPOWER/CS script functions and EMPOWER/GV functions and commands.

The *EMPOWER/CS Script Development* and *Multi-User Testing* Manuals guided you through the steps required to create and execute sophisticated load testing scripts with EMPOWER/CS. This manual provides reference information to help you understand how to use the various script functions and commands available.

The complete documentation for EMPOWER/CS includes three user's manuals which include general use information, installation instructions, technical reference material, and examples. The following list identifies each user manual:

EMPOWER/CS Script Development

Describes how to create and execute scripts to perform realistic load tests on your client/server environment (This process involves using the EMPOWER/CS tools Capture and Csccl and editing and enhancing your scripts to make them unique to your environment.)

Multi-User Testing

Describes how to use the multi-user tools Mix, Extract, Report, Draw, Monitor, and Global Variables (GV) for emulating realistic loads and measuring performance data

EMPOWER/CS Reference

Describes all commands, functions, and possible error messages for EMPOWER/CS (This manual also includes technical support information for contacting PERFORMIX, Inc.)

This *Reference Manual* is divided into the following sections:

- | | |
|------------|---|
| Section 1: | Introduces you to the EMPOWER/CS reference manual |
| Section 2: | Describes EMPOWER/CS script functions and GV commands |
| Section 3: | Lists and describes possible warning and error messages generated by EMPOWER/CS |
| Section 4: | Provides information on contacting PERFORMIX, Inc. technical support |

The conventions followed in this User Guide are listed below:

Regular Font	Used for all regular body text
Arial Font	Represents the MS Windows environment
Mono-Spaced Font	Used for all command, function, and file names; for all examples; and, generally, for any computer-generated text
Bold Mono-Spaced Font	In examples, represents entries made by the EMPOWER/CS user
<code>{-S scriptid}</code>	In command syntaxes, text within these square brackets represents optional command parameters
<code>gv_stat (name -s)</code>	Vertical lines () separate command parameter choices
...	Within scripts, the ellipsis marks indicate that some script content was left out for brevity

The term, "SUT," refers to your client/server system under test. The client/server SUT includes the database server and the database applications on that server used for your emulation.

© 2006 The Authors
Journal compilation © 2006 Blackwell Publishing Ltd

SEE ALSO Fioautorewind, Fioclose, Fioseek

EMPOWER/CS-V1.0.1

EMPOWER/CS-V1.0.1

gv_xor	-	Apply bit-wise EXCLUSIVE-OR masking to a variable
Gv_xor, Gv_xorv	-	Apply bit-wise EXCLUSIVE-OR masking to a variable
Hostname	-	Specify the name of the host machine
InitialWindow	-	Restore the Windows desktop as captured
Inote	-	Record a Monitor message (integer)
KeyDown	-	Emulate pressing a key
KeyPress	-	Emulate pressing and releasing a key
KeyUp	-	Emulate releasing a key
Language	-	Specify the language to be used
LeftButtonDown	-	Emulate pressing the left button down on the mouse
LeftButtonPress	-	Emulate pressing and releasing the left button on the mouse
LeftButtonUp	-	Emulate releasing the left button
LeftDblPress	-	Emulate two consecutive presses and releases of the left button of the mouse
Log	-	Record a message in the log file
Logoff	-	Close the communication link to the database
Logon	-	Open a communication link to the database
MiddleButtonDown	-	Emulate pressing the middle button down on the mouse
MiddleButtonPress	-	Emulate pressing and releasing the middle button on the mouse
MiddleButtonUp	-	Emulate releasing the middle button on the mouse
MiddleDblPress	-	Emulate two consecutive presses and releases of the middle button of the mouse
Note	-	Record a Monitor message (string)
Open	-	Open a cursor structure
Openenv	-	Open a database environment

EMPOWER/CS-V1.0.1

User's Guide—Reference

Unset	-	Turn script options off
Username	-	Specify the name of the user
WindowRcv	-	Paint window on screen

FUNCTION	AppName
SYNTAX	<pre>AppName(lognum, appname) int lognum; char *appname;</pre>
DESCRIPTION	<p><i>Parameters</i></p> <p>lognum The number of the log on structure that will access the SUT</p> <p>appname The name of the client application that will interact with the SUT</p> <p><i>Comments</i></p> <p>The AppName() function is inserted into the script before a log on connection when the name of the client application that will be interacting with the SUT is specified. During script execution, the AppName() function is used to define the application name a logon connection will use to interact with the SUT. Application names help the SUT to identify logon connections.</p> <p><i>Note:</i> Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.</p>
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.
SEE ALSO	Hostname, Language, Password, Servername, Username

CONVOLUTION

EXAMPLES

In the following example, the script recorded an application delay of 1.16 seconds to draw the Program Manager window.

```
AppWait(1.16);  
WindowRcv("SfDwAcSfSfSfPt");  
  
CurrentWindow("Program Manager");
```

SEE ALSO

AppWaitFactor, WindowRcv

FUNCTION	AppWaitFactor
SYNTAX	<code>AppWaitFactor(n)</code> <code>double n;</code>
DESCRIPTION	<p><i>Parameters</i></p> <p><code>n</code> The multiplying factor for the value of the AppWait delay. The default value of <code>n</code> is 1.</p> <p><i>Comments</i></p> <p>The <code>AppWaitFactor()</code> function specifies a multiplying factor for the value of the AppWait delay.</p> <p>You may insert the <code>AppWaitFactor()</code> function into a script when editing. This function applies only to <code>AppWait()</code> functions that occur after the <code>AppWaitFactor()</code> function in the script. Multiple <code>AppWaitFactor()</code> functions may be inserted into a single script.</p> <p>The <code>AppWaitFactor()</code> function is particularly useful when an increased load on the server could cause increased AppWait delays where none occurred in previous script executions.</p>
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.
EXAMPLES	The following example doubles the <code>AppWait()</code> for closing the General Ledger window, and sets the <code>AppWait()</code> back to 1 for opening the Accounting application:


```
CurrentWindow("General Ledger",20,30,234,234);

ButtonPush("Close",254,261);
AppWaitFactor(2);

AppWait(1.6);
WindowRcv("SfDwAcSfSfSfPt");

CurrentWindow("ProgramManager",0,0,1048,1048);

ButtonPush("Accounting Application",23,45);

AppWaitFactor(1);

AppWait(2.1);
WindowRcv("SfCwCwAcSfSfSfSfPt");

CurrentWindow("Accounting",20,40,234,234);
```

SEE ALSO

AppWait

Comments

RETURN VALUE (not applicable)

```
Empower();
{
AppWait(0.33);
WindowRcv("SfSfSfPt");
...

openwindow();

LeftButtonDown(132,91);
LeftButtonUp(158,212);

AppWait(3.52);
WindowRcv("ScSfSfSfPt");

Endscenario("example");
}

openwindow()
{
Beginsource("script1");
Beginfunction("openwindow");
```

EMPOWER/CS-V1.0.1


```
LeftDblPress(438,304);
LeftDblClick(438,304);

AppWait(0.24);
WindowRcv("PtCoCwCwDwAcSfCwCoSfCwCwCwSfAcPtPtPt");
WindowRcv("Pt");

CurrentWindow("New...");

/* Clicked (Button) (Cancel) */
LeftButtonDown(342,256);

Endfunction("openwindow");
Endsource();
}
```

Timestamps in the log file will correspond to the `Beginfunction()` and `Endfunction()` statements in the script.

Example:

```
>>> 12 Beginfunction("openwindow") 14:43:23.01
. . .
>>> 27 Endfunction("openwindow") 14:43:27.15
```

SEE ALSO

`Beginsource`, `Endfunction`, `Endsource`

EMPOWER/CS-V1.0.1

Endscenario

FUNCTION **Beginsource**

SYNTAX Beginsource(str)
 char *str;

DESCRIPTION *Parameters*
 str The name of the script source file which is a null-terminated
 character string

Comments

Modular script design is achieved by storing one or more functions in separate script source files, which allows a script to include a function call rather than repeating a set of interactions. When each source file is compiled with the `-c` option of the `csc` command, an object file for each script is created which can be compiled with and linked to the main script file.

If a script is compiled from multiple source files, each source file should include `Beginsource()` and `Endsource()` statements. `Beginsource()` and `Endsource()` specify the source file used for script execution.

When editing your scripts, you should insert the `Beginsource()` function at the source file's entry point, typically just before the first executable statement in each function. The `Endsource()` function should be placed at file's exit point, typically just after the last executable statement in each function.

`Beginsource()` and `Endsource()` also are placed around the C functions defined during Capture. Functions are formatted in this way so that for a multi-user emulation you may break the function out of a script into a separate source file to be used by multiple scripts.

The `Beginsource()` and `Endsource()` statements are used in Monitor to indicate the source file being executed at a certain point of script execution.

RETURN VALUE (not applicable)

EXAMPLES In the following example, elements of a function called `logoff1()` may be contained in a separate script file called `exitapp.c`, as shown:

```
logoff1()
{
  Beginsource("exitapp");
  Beginfunction("logoff1");

  Think(9.05);

  LeftButtonDown(207,97);
  LeftButtonUp(226,241);

  AppWait(0.33);
  WindowRcv("ScDwAc");

  CurrentWindow("Capture - script1",21,690,57,726);

  Commit(LOG1);

  Close(CUR1);
  Logoff(LOG1);
  Closenv(ORACLE);

  Endfunction("logoff1");
  Endsourced();
}
```

SEE ALSO `Endsource`

2025 年 1 月 1 日

When editing your script, you may change the `str` parameter to a string that is more meaningful for your emulation.

EXAMPLES In the following example, `Begintimer()` and `Endtimer()` were inserted into the script around database activity. Notice that the parameter to these functions lists the last user event, which was pressing the **OK** button in the window titled **Status**.

SEE ALSO Endtimer


```
BeginTransaction(LOG1);
Open(LOG1,CUR1);
...
Exec(CUR1);
Commit(LOG1);
```



```
SYNTAX      Bind(curnum, name, type, length)
            int curnum;
            char *name;
            int type, length;
```

DESCRIPTION	Parameters
	<p>curnum An identifier of the cursor structure of the associated SQL statement</p> <p>name The variable's placeholder name as listed in the SQL statement</p> <p>type - The variable's data type</p> <p>length The length of the variable in bytes</p>

If a SQL statement requires data to be input to the database, placeholders for input variables will be listed in the SQL statement and are indicated by leading colons. A `Bind()` function will be inserted into the script for each placeholder that is listed. During script execution, the `Bind()` function binds the placeholder to a variable that is to be passed to the database. For example, if a select-list item of the SQL statement includes a placeholder such as `:NAME`, the `Bind()` function is inserted into the script to bind `:NAME` to a variable.

The `Bind()` function is called after the `Parse()` statement and before `Exec()`.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

```
Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,  
SAL, COMM, DEPTNO FROM EMP WHERE EMPNO=:X");
```

• • •

SEE ALSO Bindp, BindDefine, Define

EMPOWER/CS-V1.0.1

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client application and SUT and therefore, break the script during execution.

SEE ALSO Bind, Bindp, Define

DESCRIPTION	Parameters
	<p>curnum An identifier of the cursor structure of the associated SQL statement</p> <p>pos A position index for the variable's placeholder as listed in the SQL statement</p> <p>type The variable's data type</p> <p>length The length of the variable</p>

This function may be inserted into the script instead of a `Bind()` function when a variable referenced in a SQL statement contains data to be input to the database. Instead of binding a placeholder name to the variable, the `Bindp()` function binds a variable's position index in a SQL statement to the variable.

`Bindp()` associates the address of a script variable with the specified select-list item, or placeholder, in the SQL statement. This placeholder is indicated by leading colons. The parameters of the `Bindp()` function identify the variable by a specific cursor number, the position index in the SQL statement of the variable's placeholder, the variable's data type, and the variable's length. The positions of the select list-items start at "1" for the first (or left-most) select-list item, "2" for the second, etc.

The `Bindp()` function must be called after the `Parse()` statement and before the `Exec()` function.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client application and SUT and therefore, break the script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example demonstrates Bindp() in a script file:

```
Parse(CUR1, "select ename from employee_table where
empno=:empno and deptno=:deptno");

...

Bindp(CUR1, 1, STRING, 20);      /* pos 1 is :empno */
Bindp(CUR1, 2, LONG, 4); /* pos 2 is :deptno */
```

SEE ALSO Bind, BindDefine, Define

FUNCTION **ButtonPush**

SYNTAX `int ButtonPush(str,x,y)`
 `char *str;`
 `unsigned int x,y;`

DESCRIPTION *Parameters*

<code>str</code>	The name of the pushbutton or the tree structure that lists the pushbutton
<code>x</code>	The on screen x coordinate of the pushbutton
<code>y</code>	The on screen y coordinate of the pushbutton

Comments

The `ButtonPush()` function is inserted in the script file during Capture to indicate that a MS Windows pushbutton was activated (such as **OK**, **Cancel**, **Yes**, **No**, etc.). This function is used during script execution in Display mode to move the mouse to activate a pushbutton defined in the parameter `str`. In Non-Display mode, this function simulates mouse movement as defined in the `x,y` parameters to allow for mouse pointer delay.

The format of the `str` parameter is designed so that EMPOWER/CS can easily locate the specified pushbutton during script execution in Display mode. This format is based on the MS Windows concept of a tree structure and may appear similar to the following:

`"Tools|#c1|#c4"`

The MS Windows tree structure is based on a heirarchy of windows where each window that is accessed from a primary, or parent, window is a child of that parent. The `str` parameter is listed right to left from child to parent window where the right-most item is the name of the button. If one of the windows or the button has no name, something like

If you wish to edit this function in your script file, you can use the Tree Tool under EMPOWER/CS Tools to determine the tree structure for a particular pushbutton.

EXAMPLES In the following example script segment, the user pushed the button OK in the current window, "Run", to open an application:

[illegible]

FUNCTION	Cancel
SYNTAX	<code>Cancel(num, opt)</code> <code>int num, opt;</code>
DESCRIPTION	<p><i>Parameters</i></p> <p><code>num</code> An identifier of the logon or cursor structure for which operations are to be cancelled</p> <p><code>opt</code> An option of either ALL or CURRENT</p> <p><i>Comments</i></p> <p>This function may appear throughout your script and is inserted into the script when database operations were cancelled for either a logon or cursor structure. For example, at some point during the Capture session, the user may have requested that the current operation of fetching a record was cancelled.</p> <p>During script execution, the <code>Cancel()</code> function cancels operations in progress for the specified structure without closing the structure. The <code>opt</code> parameter specifies an option of cancelling all operations or the current operation on the specified structure.</p> <p><i>Note:</i> Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.</p>
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.
EXAMPLES	The following example demonstrates the CURRENT option of <code>Cancel()</code> . If, during Capture, all desired data has been fetched for a query, the user may wish to cancel the query before it has completed.


```
Cancel(CUR1, CURRENT);
```

```
Cancel (LOG1, ALL);
```


BOOK REVIEW

SEE ALSO

EMPOWER/CS-V1.0.1

EMPOWER/CS-V1.0.1

SEE ALSO

EMPOWER/CS-V1.0.1

EMPOWER/CS-V1.0.1

An example of this function follows:

```
Parse(CUR1, "select ename from employee_table");

Define(CUR1, "1", STRING, 50);

Exec(CUR1);

Dbset(CUR1, FETCHSIZE, 1);
while (CmpVar(CUR1, "1", "Smith") != 0){
    Fetch(CUR1);
    GetNextRow(CUR1);
}
```

GetIntVar, GetVar, SetIntVar, SetVar

EMPOWER/CS-V1.0.1


```
Commit(LOG1);  
  
Close(CUR1);  
Logoff(LOG1);  
Closenv(ORACLE1);  
  
Endscenario("script1");
```

SEE ALSO

Rollback

FUNCTION	Continue
SYNTAX	<code>Continue(lognum, transnum)</code> <code>int lognum, transnum;</code>
DESCRIPTION	<p><i>Parameters</i></p> <p><code>lognum</code> An identifier of a logon communication structure</p> <p><code>transnum</code> An identifier of the transaction to be resumed</p> <p><i>Comments</i></p> <p>The <code>Continue()</code> function specifies an application-defined database transaction that is to be continued. This function is captured into the script when the client application instructs the database to resume execution of the paused transaction. It will be inserted after the associated <code>BeginTransaction()</code> and <code>Pause()</code> functions.</p> <p>The specified <code>transnum</code> parameter must correspond to a <code>transnum</code> specified in an associated <code>Pause()</code> function. A transaction can be continued only if no other transactions are currently running on the specified logon connection.</p>
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.
EXAMPLES	The following example demonstrates this function in a script file:

```
BeginTransaction(LOG1);  
  
....  
Pause(LOG1, TRANS1);  
  
BeginTransaction(LOG1);  
.  
....
```

(continued on following page...)

SEE ALSO [BeginTransaction](#), [Pause](#)

EMPOWER/CS-V1.0.1

RETURN VALUE (not applicable)

```
AppWait(4.34);
WindowRcv("SfAcPtSfPtPtSfDwAcSfSfSfPt");

CurrentWindow("Program Manager",413,679,1029,771);

KeyPress(VK_CONTROL);
KeyPress(VK_F12);

LeftButtonPress(443,112);
```

SEE ALSO InitialWindow

EMPOWER/CS-V1.0.1

must be string pointers. The string conversion specification is provided by the characters %s.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following example demonstrates a SQL statement and its corresponding data line in a script:

```
/* insert data into database */
Parse(CUR1, "insert empno, ename, empjob into emp_table");

Bind(CUR1, "empno", INT, 4);
Bind(CUR1, "ename", STRING, 30);
Bind(CUR1, "empjob", STRING, 20);

/* 123 refers to empno, Smith -- ename, typist -- empjob */
Data(CUR1, "123|Smith|typist");

Exec(CUR1);
```

The parameter to Data(), "123|Smith|typist" represents the data that was inserted into the database. 123 would correspond to EMPNO, Smith would correspond to ENAME, and typist would refer to EMPJOB.

The following example demonstrates using this function to accept a variable argument:

```
Parse(CURL, "select * from emp_table where lname = :name and empjob = :job");

Bind(CURL, ":name", STRING, 30);
Bind(CURL, ":job", STRING, 30);

/* this Data() will get name from file and alway use job of typist */
Data(CURL, "%s|typist", Fioreadfield(datafile));

Exec(CURL);
```


EMPOWER/CS-V1.0.1

SEE ALSO Timeout

Comments

The `Dbset()` options that commonly will appear in your scripts are listed below:

Specifies the number of records to fetch from the database when a `Fetch()` function is executed. This number will be less than or equal to the value specified in the `MAXARRSIZE`. The `Dbset()` function that sets the `FETCHSIZE` will occur before a `Fetch()`. The `FETCHSIZE` value can not be larger than the `MAXARRSIZE`. If the `FETCHSIZE` was specified as 50 and `MAXARRSIZE` was specified as 20, `EMPOWER/CS` will reduce the `FETCHSIZE` to 20.

Sets the maximum array size for retrieving or inserting records. If the array size is set at 20, 20 rows of data can be inserted or fetched. This option set at 20 allocates 20 placeholders for inserting or fetching 20 rows of data. The `Dbset()` function that sets the `MAXARRSIZE` will occur before a the `Bind()` and `Define()` functions in a script.

Specifies the number of rows of data to be inserted into the database. This option will be listed before the `Data()` function in a script in the format `Dbset(CUR1, INSERTSIZE, n)`. This option allows array binding. For example, if `n` is 0 or 1, one row can be inserted at a time into the database. If `n` is 50, 50 rows will be inserted into the database and 50 data lines will be listed for every row before `Exec()`. The `INSERTSIZE` value can not be larger than the specified `MAXARRSIZE`.

Specifies the offset for inserting records in an array. This option is used with the `INSERTSIZE` option. If an array includes four names and the `OFFSET` is set to 2, then inserting rows will start from the second position with the second name. The `Dbset()` function specifying this option will occur before an `Exec()` function. If the `OFFSET` is specified as larger than the `MAXARRSIZE`, a database error will occur.

Specifies whether or not to wait for resources. If this option is set to `TRUE`, the script will wait indefinitely for requested information from the database. If it is set to `FALSE` and the script does not receive the requested information, the script will receive an error that the resource was not available. Script execution will then either continue or exit based on the condition set in `Dberror()`.

Specifies whether or not to defer the `Parse()` statement. If this option is set to `TRUE`, a deferred parse will be performed when the script encounters the `Parse()` function. If the option is set to `FALSE`, a normal `Parse()` is executed.

Normally, the SQL statement is sent to the database when the script encounters `Parse()` and is processed to ensure it is semantically correct. The SQL statement is stored waiting for the `Exec()` call that actually will execute it. If `DEFER` is set to true, the SQL statement is not sent to the database until the script encounters an operation that requires input from the database such as `Exec()` or `Describe()`.

Other `Dbset()` options may appear in your scripts. A full list of all possible options follows. This list is divided into those options that set integer values and those that require `TRUE` or `FALSE` values:

Integer Value

<u>Option</u>	<u>Description</u>
<code>FETCHSIZE</code>	number of rows to be fetched
<code>MAXARRSIZE</code>	max number of rows to be fetched
<code>INSERTSIZE</code>	number of rows to be inserted
<code>OFFSET</code>	row number where to start inserting
<code>MAXCONNECT</code>	maximum number of connections in an environment
<code>PACKETSIZE</code>	maximum packet size on log
<code>ROWCOUNT</code>	maximum number of rows to return
<code>TEXTSIZE</code>	limits size of text or image data
<code>ISOLATIONLEVEL</code>	transaction isolation level
<code>AUTHOFF</code>	turns specified authorization off
<code>AUTHON</code>	turns specified authorization on
<code>CURREAD</code>	security label spec. cur read level
<code>CURWRITE</code>	security label spec. cur write lev
<code>DATEFIRST</code>	which day of week is first
<code>DATEFORMAT</code>	format of date
<code>IDENTITYOFF</code>	disable inserts into table ident column
<code>IDENTITYON</code>	enable inserts into table ident column

TRUE or FALSE

<u>Option</u>	<u>Description</u>
<code>DEFER</code>	deferred parse
<code>UPDATE</code>	cursor is for updating
<code>READONLY</code>	cursor is read only
<code>SAVEOPTS</code>	do not deallocate the structure
<code>FORCELOGOFF</code>	force logoff

[illegible]

EXAMPLES In the following example, the client application specified that the DEFER option be set to TRUE so that a deferred Parse() will occur:

In this next example, the `MAXARRSIZE` is set to a size 64 array for the subsequent script operations:

```
Dbset(CUR1, MAXARRSIZE, 64);
Define(CUR1, "1", STRING, 40);
Define(CUR1, "2", CHAR, 21);
Define(CUR1, "3", CHAR, 21);
Define(CUR1, "4", CHAR, 21);
Define(CUR1, "5", CHAR, 21);
Define(CUR1, "6", CHAR, 21);
Define(CUR1, "7", CHAR, 16);
Define(CUR1, "8", CHAR, 16);
Define(CUR1, "9", STRING, 40);
Define(CUR1, "10", STRING, 40);
Define(CUR1, "11", CHAR, 241);
Exec(CUR1);
```


EMPOWER/CS-V1.0.1

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

```

Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO
FROM EMP, UPDATE OF EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM,
DEPTNO");

Define(CUR1, "1", CHAR, 5);          /* pos1 = EMPNO */
Define(CUR1, "2", STRING, 11);       /* pos2 = ENAME */
Define(CUR1, "3", STRING, 10);       /* pos3 = JOB */
Define(CUR1, "4", STRING, 5);        /* pos4 = MGR */
Define(CUR1, "5", STRING, 10);       /* pos5 = HIREDATE */
Define(CUR1, "6", LONG, 4);          /* pos6 = SAL */
Define(CUR1, "7", STRING, 9);        /* pos7 = COMM */
Define(CUR1, "8", INT, 4);           /* pos8 = DEPTNO */

Exec(CUR1);

```

EMPOWER/CS-V1.0.1

EMPOWER/CS-V1.0.1

The following example demonstrates how `Describe()` would appear in a script:

```
Parse(CUR1, " select ename from emp_table");
Describe(CUR1, 1); /* describes ename */
```

The following is an example of the output for `Describe()` which is recorded in the script's log file:

```
>>> Describe(CUR1, 1);
size=20, type=CHAR, name=ename, namelen=10, dsize=0, prec=0,
scale=0, nullok=0
```

SEE ALSO

DescribeAll, DescribeProc

FUNCTION	DescribeAll
SYNTAX	<code>DescribeAll(curnum, pos1, pos2)</code> <code>int curnum, pos1, pos2;</code>
DESCRIPTION	<p><i>Parameters</i></p> <p><code>curnum</code> An identifier of a cursor structure of the associated SQL statement</p> <p><code>pos1</code> The starting position in the SQL statement for a Describe operation</p> <p><code>pos2</code> The ending position in the SQL statement for a Describe operation</p> <p><i>Comments</i></p> <p>A <code>DescribeAll()</code> function is inserted into the script when a <code>Describe()</code> operation was performed for each select-list item specified between two positions of the SQL statement. This function describes each select list item, or variable, starting from the position specified in <code>pos1</code> to the position specified in <code>pos2</code>.</p> <p><i>Note:</i> Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the behavior of the application and SUT and therefore, break your script during execution.</p>
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.
EXAMPLES	In the following example, the variables listed in the SQL statement from <code>CUR1</code> , starting from the first position to the twelfth position, which will be described:


```
DescribeAll(CUR1, 1, 12);
```

Describe, DescribeProc

EMPOWER/CS-V1.0.1

FUNCTION	Difftime
1	0.000000
2	0.000000
3	0.000000
4	0.000000
5	0.000000
6	0.000000
7	0.000000
8	0.000000
9	0.000000
10	0.000000
11	0.000000
12	0.000000
13	0.000000
14	0.000000
15	0.000000
16	0.000000
17	0.000000
18	0.000000
19	0.000000
20	0.000000
21	0.000000
22	0.000000
23	0.000000
24	0.000000
25	0.000000
26	0.000000
27	0.000000
28	0.000000
29	0.000000
30	0.000000
31	0.000000
32	0.000000
33	0.000000
34	0.000000
35	0.000000
36	0.000000
37	0.000000
38	0.000000
39	0.000000
40	0.000000
41	0.000000
42	0.000000
43	0.000000
44	0.000000
45	0.000000
46	0.000000
47	0.000000
48	0.000000
49	0.000000
50	0.000000
51	0.000000
52	0.000000
53	0.000000
54	0.000000
55	0.000000
56	0.000000
57	0.000000
58	0.000000
59	0.000000
60	0.000000
61	0.000000
62	0.000000
63	0.000000
64	0.000000
65	0.000000
66	0.000000
67	0.000000
68	0.000000
69	0.000000
70	0.000000
71	0.000000
72	0.000000
73	0.000000
74	0.000000
75	0.000000
76	0.000000
77	0.000000
78	0.000000
79	0.000000
80	0.000000
81	0.000000
82	0.000000
83	0.000000
84	0.000000
85	0.000000
86	0.000000
87	0.000000
88	0.000000
89	0.000000
90	0.000000
91	0.000000
92	0.000000
93	0.000000
94	0.000000
95	0.000000
96	0.000000
97	0.000000
98	0.000000
99	0.000000
100	0.000000

```
SYNTAX      double Diffftime(first, second, diff);
            struct timevalue *first, *second, *diff;
```

DESCRIPTION	Parameters
-------------	------------

first	The first time stamp value
-------	----------------------------

second The second time stamp value

diff	Variable that stores the difference between first and second
------	--

Comments

You may insert this function into your script when editing. It is used to ensure that certain activities in a script occur in a specified amount of time.

`DiffTime()` computes the difference in seconds between the `first` and `second` values and stores the result in the variable `diff`. The time difference is returned in a double value. This allows a time difference to be expressed as a floating point (fractional) number, as in 1.5 (one and a half seconds). Normally, the first value is earlier than the second.

The first, second, and diff values are time stamps of struct timeval type. The variable struct timeval is defined in \$EMPOWER/h/empower.h as:

```
struct timeval {
    long sec;    /* seconds since Jan 1. 1970 */
    short hsec; /* and hundredths of a second */
}
```

RETURN VALUE The return value is positive if the first time value is earlier than the second time value. Otherwise, the return value is negative.

EXAMPLES

This example script segment measures the time it takes to enter information into a field and press the left mouse button. One of the past uses of the `Difftime()` function was to help pace a script to provide a required transaction throughput. This operation now is accomplished with Empower's Pace functions.

```
char buf[10];
double diffm;
struct timevalue time1, time2;

...

Time(&time1);
Type("1234^M");

LeftButtonPress(282,174);

AppWait(0.05);
WindowRcv("SfSfSfSf");
Time(&time2);

diffm=Difftime(&time1, &time2, 0);
sprintf(buf, "time is %.2f", diffm);
Log(buf);
```

SEE ALSO

Time, Eventtime, Paceconstant, Pacetne, Paceuniform

FUNCTION	Endfunction
SYNTAX	Endfunction(str) char *str;
DESCRIPTION	<p><i>Parameters</i></p> <p>str The name of the function. This parameter is a null-terminated string.</p> <p><i>Comments</i></p> <p>Endfunction() is used to mark the end of a function. This function is inserted into the script automatically during Capture when you specify the end of a function or, it can be inserted when editing your script.</p> <p>Endfunction() works with and must have a corresponding Beginfunction() to define a task you wish to measure. The Endfunction() statement and its corresponding Beginfunction() must use the same function name, i.e. str parameter.</p> <p>Endfunction() records the name of the function and the time at which the event occurred in the log file.</p>
RETURN VALUE	(not applicable)
EXAMPLES	<p>In the following example, the Beginfunction() and Endfunction() statements are placed in a C language function that consists of opening a window. The name of the function is openwindow. A function call is placed within the scenario and the actual function is listed after Endscenario():</p>


```
Empower();
{
AppWait(0.33);
WindowRcv("SfSfSfPt");

...

openwindow();

LeftButtonDown(132,91);
LeftButtonUp(158,212);

AppWait(3.52);
WindowRcv("ScSfSfSfPt");

Endscenario("example");
}

openwindow()
{
Beginsource("example");
Beginfunction("openwindow");

LeftDbPress(438,304);
LeftDbClick(438,304);

AppWait(0.24);
WindowRcv("PtCoCwCwDwAcSfCwCoSfCwCwCwCwCwCwCwSfAcPtPtPtPtPt");
WindowRcv("Pt");

CurrentWindow("New...");

/* Clicked (Button) (Cancel) */
LeftButtonDown(342,256);

Endfunction("openwindow");
Endsource();
}
```



```
>>> 26 Endfunction("query1") 04:11:23.29
```

SEE ALSO

Beginfunction

FUNCTION **Endscenario**

SYNTAX `Endscenario(str)`
 `char *str;`

DESCRIPTION *Parameters*

`str` The name of the script as defined in the Capture session.
 This name is a null-terminated character string.

Comments

This function is inserted automatically into your script file during Capture to define the end of a scenario. `Endscenario()` works with and must have a corresponding `Beginscenario()` to define a scenario. The `Endscenario()` function and its corresponding `Beginscenario()` must use the same scenario name, i.e., `str` parameter.

`Endscenario()` will record in the log file the name of the scenario and the time at which the event occurred.

EMPOWER/CS provides summary performance information for each scenario. For example, the Report tool provides scenario start and stop times, duration, throughput, and average response times.

By default, the duration of the test is determined by the time stamps of the first `Beginscenario()` and of the last `Endscenario()`.

RETURN VALUE (not applicable)

EXAMPLES The name of the scenario is taken from the name of the script source file. Initiation of capturing the script `example` would cause the following functions to be inserted at the beginning and end of the script:

```
Beginscenario("example")
...
Endscenario("example")
```


The log file created from script execution will contain time stamps for the beginning and end of the scenario.

Example:

```
>>> 10 Beginscenario("example") 12:05:29.00
...
>>> 462 Endscenario("example") 12:08:23.07
```

SEE ALSO

Beginscenario


```
logoff1()  
{  
  Beginsource("exitapp");  
  Beginfunction("logoff1");  
  
  Think(9.05);  
  
  LeftButtonDown(207,97);  
  LeftButtonUp(226,241);  
  
  AppWait(0.33);  
  WindowRcv("ScDwAc");  
}
```

(continued on following page...)

SEE ALSO

EMPOWER/CS-V1.0.1

Comments

The `BeginTimer()` and `EndTimer()` functions can be nested.

The `Endtimer()` function will occur at the end of the database activity when a window is drawn on screen or another set of user input begins.

EMPOWER/CS-V1.0.1

When editing your script, you may change the `str` parameter to a string that is more meaningful for your emulation.

EXAMPLES In the following example, `Beginntimer()` and `Endntimer()` were inserted into the script around database activity. Notice that the parameter to these functions lists the last user event, which was pressing the **OK** button in the window titled **Status**.

```
CurrentWindow("Status",240,180,408,301);
ButtonPush("OK",322,267);

WindowRcv("SfAcSfDw");

...

Beginntimer("Status_OK");

Close(CUR1);
Logoff(LOG1);
Closenv(ORACLE);

Endtimer("Status_OK");
```

SEE ALSO Begintimer

FUNCTION **Eventtime**

SYNTAX `int Eventtime(event, p)`
 `int event;`
 `struct timevalue *p;`

DESCRIPTION *Parameters*

event	The name of an EMPOWER/CS event
p	A timevalue structure where the returned event time is stored

Comments

You can insert this function when editing your script file. `Eventtime()` retrieves the time at which the last of several events occurred. Valid events defined in `$EMPOWER/h/empower.h` are:

<u>EVENT</u>	<u>DESCRIPTION</u>
TSTART	time that script starts
TBF	time at last beginfunction
TEF	time at last endfunction
TBS	time at last beginscenario
TES	time at last endscenario
TBT	time at last begintimer
TET	time at last endtimer

The returned event time is stored in a timevalue structure pointed to by `p`. The struct timevalue is defined in `$EMPOWER/h/empowercs.h` as:

```
struct timevalue {
    long  sec;    /*seconds since Jan. 1 1970*/
    short hsec;  /*hundredths of a second*/
}
```


EXAMPLES: This example script segment uses `Eventtime()` and `DiffTime()` to record the duration of events in the log file. Before the EMPOWER Pace functions were developed, these functions helped the user to maintain a constant transaction throughput. (Some script content is left out for brevity.)

```
char buf[30];
double difftm;
struct timevalue time1, time2, time3, time4;

Beginscenario("script1");

...

Beginfunction("logout1");

...

Endfunction("logout1");

Endscenario("script1");

Eventtime(TBF, &time1);
Eventtime(TEF, &time2);
Eventtime(TBS, &time3);
Eventtime(TES, &time4);

difftm=Difftime(&time1,&time2,0);
sprintf(buf, "logout1 function was %.2f seconds", difftm);
Log(buf);

difftm=Difftime(&time3,&time4,0);
sprintf(buf, "scenario duration was %.2f seconds", difftm);
Log(buf);
```



```
>>> 20 Beginscenario("script1") 14:10:44.26
...
>>> 368 Beginfunction("logout1") 14:11:48.20
...
>>> 390 Endfunction("logout1") 14:11:54.37
>>> 352 Endscenario("script1") 14:11:54.37
>>> 359 Log("logout1 function was 6.17 seconds")
>>> 362 Log("scenario duration was 70.11 seconds")
```

Difftime, Paceconstant, Pacetne, Paceuniform, Time

FUNCTION	Exec
SYNTAX	<code>Exec(curnum)</code> <code>int curnum;</code>
DESCRIPTION	<p><i>Parameters</i></p> <p><code>curnum</code> An identifier of the cursor communication structure specified in the associated <code>Parse()</code></p> <p><i>Comments</i></p> <p>This function is inserted into the script when the <code>Parse()</code> statement is executed on the SUT. During script execution, <code>Exec()</code> instructs the SUT to execute the current <code>Parse()</code> statement. <code>Exec()</code> also forces all data and other relevant information to be passed from the client to the SUT.</p> <p><code>Exec()</code> is called after a <code>Parse()</code> and all the <code>Bind()</code> or <code>Define()</code> statements associated with a specific cursor. For database queries, the requested rows are fetched with the <code>Fetch()</code> function after <code>Exec()</code> has been called.</p> <p><i>Note:</i> Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the application and SUT and therefore, break the script during execution.</p>
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.
EXAMPLES	In the following script segment the <code>Exec()</code> statement is called for the cursor, <code>CUR1</code> :

SEE ALSO

EMPOWER/CS-V1.0.1

2025

edit this function or remove it from the script. If you change this function from when it was captured, you may drastically alter the expected behavior of the application and SUT and therefore, break the script during execution.

RETURN VALUE `Fetch()` returns the number of rows fetched from the database. If no more rows can be fetched, an error will be returned.

EXAMPLES In the following example, the `FETCHSIZE` for the cursor, `CUR1`, is set to 4 in the `Dbset()` function:

```
Dbset(CUR1, FETCHSIZE 4)
```

In the following script segment, `Fetch()` is called for the cursor, `CUR1`:

```
Parse(CUR1, " SELECT ID, FIRST_NAME, LAST_NAME, ADDRESS_LINE_1,
ADDRESS_LINE_2, ADDRESS_LINE_3, PHONE_NUMBER, FAX_NUMBER, COMM_PAID_YTD,
ACCOUNT_BALANCE, COMMENTS FROM CUSTOMERS ");

DescribeAll(CUR1, 1, 12);

Dbset(CUR1, MAXARRSIZE, 64);
Define(CUR1, "1", STRING, 40);
Define(CUR1, "2", CHAR, 21);
Define(CUR1, "3", CHAR, 21);
Define(CUR1, "4", CHAR, 21);
Define(CUR1, "5", CHAR, 21);
Define(CUR1, "6", CHAR, 21);
Define(CUR1, "7", CHAR, 16);
Define(CUR1, "8", CHAR, 16);
Define(CUR1, "9", STRING, 40);
Define(CUR1, "10", STRING, 40);
Define(CUR1, "11", CHAR, 241);
Exec(CUR1);

Dbset(CUR1, FETCHSIZE, 64);
Fetch(CUR1);
```


SEE ALSO Dbset, FetchRaw, GetNextRow, Parse

Figure 1

EMPOWER/CS-V1.0.1

The following example demonstrates a `FetchRaw()` function that fetches binary data in 1024 byte sections until no more data can be fetched:

SEE ALSO Fetch

File Input/Output Functions

You can insert EMPOWER/CS File Input/Output functions into your script when editing. These functions are used to read and write files. Such capabilities are useful for load tests requiring interaction with data files on the UNIX driver machine and for simplifying complex scripts such as database entry scripts.

The EMPOWER/CS file input/output functions are used in your scripts to read data from a file, send data from the file to the SUT, receive data from the SUT, and write those data to a file. These functions simplify the C language statements that would need to be added to scripts to accomplish the same thing.

The environment variable `E_FIOPATH` can be used to specify the directory in which the files to be accessed reside. A file must be opened before it can be accessed with the file input/output functions. If a file contains NULL characters, an error will occur when the file is read by an input/output function.

Three global variables are used for file input/output. They are defined automatically as follows:

```
unsigned char *FIOBUFFER
int FIOLEN
int FIOBUFFERSZ
```

The variable `FIOBUFFER` is a pointer to the characters read from the file. This variable often is used when sending data read from a file to the SUT. The variable `FIOLEN` is the number of valid characters in `FIOBUFFER`. If the value of `FIOLEN` is less than or equal to zero, then either an error occurred or the end-of-file (EOF) was reached. The variable `FIOBUFFERSZ` is the maximum size of the data that can be read at one time. The default value of `FIOBUFFERSZ` is 512 characters. If the value of `FIOBUFFERSZ` is redefined in a script, it must be redefined before any file input/output functions that reference the file are encountered.

These functions are described in the following reference entries.


```
Fioautorewind("info");
Fioreadline("info");
Type("%s", FIOBUFFER);
```

EMPOWER/CS-V1.0.1

SEE ALSO

EMPOWER/CS-V1.0.1

EMPOWER/CS-V1.0.1


```
LeftButtonPress(360,211);
```

```
Fioopen("numbers", "r");
Fioreadline("numbers");
Type("%s", FIOBUFFER);
Fioclose("numbers");
```

Fioopen

FUNCTION	Fiodelimiter				
SYNTAX	<pre>Fiodelimiter(filename, delimiters) char *filename; unsigned char *delimiters;</pre>				
DESCRIPTION	<p><i>Parameters</i></p> <table><tr><td>filename</td><td>The file for the operation</td></tr><tr><td>delimiters</td><td>The field delimiters for the specified file</td></tr></table> <p><i>Comments</i></p> <p>See the description under File Input/Output Functions for a general explanation of these functions.</p> <p>The <code>Fiodelimiter()</code> function defines the field delimiters for the file <code>filename</code>. The default is <code>"\t\n"</code> where <code>"\n"</code> is always a delimiter (<code>"\t"</code> is tab and <code>"\n"</code> is new line or linefeed).</p>	filename	The file for the operation	delimiters	The field delimiters for the specified file
filename	The file for the operation				
delimiters	The field delimiters for the specified file				
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.				
EXAMPLES	<p>The following example illustrates using the functions <code>Fioreadfield()</code> and <code>Fioreadfields()</code> to read one or more fields from a file. <code>Fiodelimiter()</code> specifies that the field delimiter in the file <code>inputfile</code> is a comma. (Some script content was left out for brevity.)</p>				

The following is a portion of the file inputfile:

SEE ALSO

EMPOWER/CS-V1.0.1


```
SYNTAX      Fioopen(filename, mode)
            char *filename, *mode;
```

Comments

The `Fioopen()` function opens the file `filename`. The parameter `mode` specifies how the file is opened. The following list demonstrates how the file is opened by specifying different `mode` parameters:

<u>Mode</u>	<u>How File Is Opened</u>
r	Opened at the beginning for reading only
w	Truncated or created for writing only
a	Opened at the end for writing only
r+	Opened at the beginning for reading and writing
w+	Truncated or created for reading or writing
a+	Opened at the end for reading and writing

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following script opens the "names" file, reads a line from the file, and transmits the name to the SUT with the `Type()` function. Then it closes the "names" file, opens the "numbers" file, reads a line from this


```
Fioopen("names", "r");
Fioreadline("names");
Type("%s", FIOBUFFER);
Fioclose("names");

LeftButtonPress(360,211);

AppWait(0.06);
WindowRcv("SfSfPt");

Fioopen("numbers", "r");
Fioreadline("numbers");
Type("%s", FIOBUFFER);
Fioclose("numbers");
```

Fioclose


```
Fioreadchar("letters", 10);
Type("%s", FIOBUFFER);
```


EMPOWER/CS-V1.0.1

The following is a portion of the file inputfile:

SEE ALSO Fioclose, Fiodelimiter, Fiopen, Fioreadchar, Fioreadfields, Fioreadline

EMPOWER/CS-V1.0.1

The following is a portion of the file inputfile:

SEE ALSO

EMPOWER/CS-V1.0.1


```
Fioopen("names", "r");
Fioreadline("names");
Type("%s", FIOBUFFER);
Fioclose("names");

LeftButtonPress(360,211);
```

(continued on following page...)

SEE ALSO Fioclose, Fioopen, Fioreadchar, Fioreadfield, Fioreadfields


```
Fioautorewind("info");
Fioreadline("info");
Type("%s", FIOBUFFER);
```

EMPOWER/CS-V1.0.1

SEE ALSO Fioautorewind, Fioclose, Fioseek

EMPOWER/CS-V1.0.1

Introduction

\$ fioshare cmds

```
Fioshare("cmds");

Fioreadline("cmds");
if (FIOLEN == -1)
{
    Log("end of the date file");
    exit(1);
}
Type(FIOBUFFER, "^M", "");
}
```

```
$ fioshare cmds
$ ls -i cmds
59449 cmds
$ gv_stat
gv_stat:  EMPOWER/GV V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95
Name          Type          Value Allocated  Protector
-----
59449         long int         0              0
```

Fioclose, Fiounshare, gv_stat

EMPOWER/CS-V1.0.1

EMPOWER/CS-V1.0.1

Copyright PERFORMIX, Inc. © 1995

EMPOWER/CS-V1.0.1

FUNCTION	Fiowritechar
SYNTAX	<pre>Fiowritechar(filename, buf, n) char *filename, *buf; long n;</pre>
DESCRIPTION	<p><i>Parameters</i></p> <p>filename The file to be used for the operation</p> <p>buf The file buffer</p> <p>n The number of bytes from the buffer</p> <p><i>Comments</i></p> <p>See the description under File Input/Output Functions for a general explanation of these functions.</p> <p>The <code>Fiowritechar()</code> function writes <code>n</code> bytes from the buffer, <code>buf</code>, to the file <code>filename</code>. If the file is not currently open, it automatically is created or truncated and opened for reading and writing.</p>
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.
SEE ALSO	<code>Fioclose</code> , <code>Fioopen</code>

THE UNIVERSITY OF CHICAGO

SEE ALSO

EMPOWER/CS-V1.0.1

FUNCTION **GetIntVar**

SYNTAX `GetIntVar(curnum, var)`
 `int curnum;`
 `char *var;`

DESCRIPTION *Parameters*
 `curnum` A cursor communication structure

 `var` The name of the variable listed in the `Parse()` statement

 Comments
 You can insert `GetIntVar()` into your script file to return the current value of the specified variable that was listed in the `Parse()` statement.

 This function should be inserted into the script after the `Fetch()` and `GetNextRow()` functions.

RETURN VALUE This function returns an integer for the current value of the variable.

EXAMPLES The following example demonstrates using `GetIntVar()` within a script:

```
int empno;

...

Parse(CUR1, "select ename, empno from employee_table");

Define(CUR1, "1", STRING, 50);
Define(CUR1, "2", INT, 4);

Exec(CUR1);
```

(continued on the following page...)

4

CmpVar, GetVar, SetIntVar, SetVar


```
char *empname, *empno;

...

Parse(CUR1, "select ename, empno from employee_table");

Define(CUR1, "1", STRING, 50);
Define(CUR1, "2", INT, 4);

Exec(CUR1);
```

(continued on following page...)

SEE ALSO

EMPOWER/CS-V1.0.1

DESCRIPTION	The <code>gv_add</code> command is entered at the command line and updates the value of a specified variable by adding a specified amount to the variable's current value. The parameter <code>value</code> is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.
-------------	---

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

```
$ gv_add users 4
5
$ gv_read users
9
```

EMPOWER/CS-V1.0.1

FUNCTION **Gv_add, Gv_addv**

SYNTAX

```
int Gv_add(name, value)
char *name;
int value;

int Gv_addv(name, value, oldvalue)
char *name;
```

DESCRIPTION *Parameters*

name The name of the global variable

value The operand for the operation

oldvalue The pointer location where the original value should be stored

Comments

The `Gv_add` function updates the value of a specified variable by adding a specified amount to the current value of a variable. You can insert these functions into your script when editing the script file.

`Gv_add()` is used if the variable is an integer, and `Gv_addv()` is used if the variable is not an integer.

RETURN VALUE `Gv_add()` returns the original value of the specified variable.

`Gv_addv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

EXAMPLES

The following example demonstrates using `Gv_add()` in a script file. In this example, if the variable `customer` is equal to zero, then 2 will be added to the current value of the variable `users`.


```
if (customer == 0)
    Gv_add(users, 2);
```

SEE ALSO [gv_add](#)

2025

FUNCTION **Gv_alloc**

SYNTAX `void Gv_alloc(name, type)`
 `char *name, *type;`

DESCRIPTION *Parameters*

<code>name</code>	The name of the global variable
<code>type</code>	The global variable type

Comments

The `Gv_alloc()` function allocates the script's access to the specified global variable. Access should be allocated for a variable so that a script can use the variable in subsequent Global Variable functions. The `Gv_alloc()` function should be the first function in the script if the script references a global variable.

An error will result if the `name` and `type` parameters of the `Gv_alloc()` function do not match the actual name and type of the variable specified when the variable was created with the `gv_init` command.

An error will result if the specified global variable does not exist, if the specified variable type does not match the global variable type, or if the global variable has already been allocated to the script.

If an error occurs, the script exits and an error message is sent to the standard error destination.

RETURN VALUE (not applicable)

EXAMPLES To allocate a script's access to the variable `users` which is an integer type, the following function must be included in the script file:

```
Gv_alloc("users", "int");
```



```
$ gv_init user int 50
```

Gv_free, gv_init

DESCRIPTION	The <code>gv_and</code> command is entered at the shell prompt and updates the value of a specified variable by applying a bit-wise AND masking operation to the variable. The parameter <code>value</code> is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.
-------------	---

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_and

Copyright PERFORMIX, Inc. © 1995

COMMAND `gv_dec`

SYNTAX gv_dec name

DESCRIPTION	The <code>gv_dec</code> command is entered at the shell prompt and decreases the value of the specified variable by one. When the <code>gv_dec</code> command is entered, the original value is written to the standard output destination before the new, decremented value is assigned.
-------------	---

RETURN CODE	If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.
-------------	--

EXAMPLES In the following example the current value of the variable `count` is 5.
To decrease this value by one use the following `gv_dec` command.
Then, use the `gv_read` command to get the new value of the variable:

```
$ gv_dec count
5
$ gv_read count
4
```

SEE ALSO Gv_dec, Gv_inc, gv_inc


```
if(Gv_dec("amount") == 0)
    close_account();
```


COMMAND **gv_div**

SYNTAX **gv_div** name value

DESCRIPTION The **gv_div** command is entered at the shell prompt and updates the value of a specified variable by dividing the variable's current value by a specified amount. The parameter **value** is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example, suppose the variable **users** has a current value of 8 and you wish to change the value by dividing it by 4. The interaction would be as follows:

```
$ gv_div users 4
8
$ gv_read users
2
```

SEE ALSO **Gv_div**

FUNCTION	Gv_div, Gv_divv
SYNTAX	<pre>int Gv_div(name, value) char *name; int value; int Gv_divv(name, value, oldvalue) char *name;</pre>
DESCRIPTION	<p><i>Parameters</i></p> <p>name The name of the global variable</p> <p>value The operand for the operation</p> <p>oldvalue The pointer location where the original value should be stored (for Gv_divv())</p> <p><i>Comments</i></p> <p>The Gv_div() and Gv_divv() functions update the value of a specified variable by dividing the current value of the variable by a specified amount. You can insert this function into your script when editing the script file.</p> <p>Gv_div() is used if the variable is an integer, and Gv_divv() is used if the variable is not an integer.</p>
RETURN VALUE	Gv_div() returns the original value of the specified variable and Gv_divv() copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.
SEE ALSO	gv_div


```
int number;
...
Gv_alloc("amount");
number=Gv_read("amount");
if (number > 10)
    order();
Gv_free("amount");
```


Gv_allocate

SEE ALSO [Gv_getparallel](#), [Gv_parallel](#), [Gv_setparallel](#), [Gv_unparallel](#), [gv_parallel](#), [gv_setparallel](#), [gv_unparallel](#)

EMPOWER/CS-V1.0.1

SEE ALSO Gv_dec, Gv_inc, gv_dec

EMPOWER/CS-V1.0.1

To increment the non-integer variable `balance`, the following example can be used:

```
float curbalance;  
Gv_alloc("balance", "float");  
Gv_incv("balance", &curbalance);
```

SEE ALSO `Gv_dec`, `Gv_decv`, `gv_dec`, `gv_inc`

COMMAND	<code>gv_init</code>
SYNTAX	<code>gv_init name [type] value</code>
DESCRIPTION	<p>The <code>gv_init</code> command is entered at the shell prompt and is used to initialize a variable. If the variable does not exist, it is created with the specified type and initial value. If the variable exists when the <code>gv_init</code> command is entered, the variable is reset to the specified value.</p> <p>The parameter <code>type</code> is required if the variable does not exist. If the variable exists, the parameter <code>type</code> is optional and the type and value specified in <code>gv_init</code> must correspond to the existing variable type. If a different type is specified or if the new value specified does not correspond to the existing type, the command fails.</p> <p>The default maximum number of variables is 128. The maximum length of a variable name is 14 characters. The maximum length of the value of a string variable is 32 characters.</p>
RETURN CODE	If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.
EXAMPLES	<p>The <code>gv_init</code> command creates a variable with a specified name, variable type, and initial value. In the following example, <code>gv_init</code> is the Global Variable command, <code>customer</code> is the variable name, <code>int</code> specifies that <code>customer</code> is an integer, and 100 is the initial value of the variable <code>customer</code>.</p>

```
$ gv_init customer int 100
```

To specify the same variable as a parallel variable, you would enter the following:

```
$ gv_init customer parallel 100
```


To use the following variable users during script execution:

The following function should be included in the script:

SEE ALSO

Gv_alloc, gv_seg

DESCRIPTION	<p>The <code>gv_lshift</code> command is entered at the shell prompt and updates the value of a specified variable by performing a bit-wise shift to the left on a variable. The parameter <code>value</code> is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.</p>
-------------	---

RETURN CODE	If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.
-------------	--

SEE ALSO Gv_lshift, Gv_rshift


```
SYNTAX      int Gv_lshift(name, value)
            char *name;
            int value;

            int Gv_lshiftv(name, value, oldvalue)
            char *name;
```

DESCRIPTION	<i>Parameters</i>
	name The name of the global variable
	value The operand for the operation
	oldvalue The pointer location where the original value should be stored (for Gv_lshiftv())
	<i>Comments</i>
	These functions are used with the EMPOWER/GV tool and are inserted into your script file when editing.

The `Gv_lshift()` and `Gv_lshiftnv()` functions update the value of a specified variable by performing a bit-wise shift to the left on the variable.

Gv_lshift() is used if the variable is an integer, and Gv_lshiftnv() is used if the variable is not an integer.

RETURN VALUE `Gv_lshift()` returns the original value of the specified variable and `Gv_lshiftpv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

SEE ALSO [Gv_rshift](#), [Gv_rshiftv](#), [gv_lshift](#), [gv_rshift](#)


```
$ gv_init count int 10
$ gv_mod count 3
10
$ gv_read count
1
```



```
$ gv_init count int 10
$ gv_mod count 2
10
$ gr_read count
0
```

Gv_mod, Gv_modv

EMPOWER/CS-V1.0.1

DESCRIPTION	The <code>gv_mul</code> command is entered at the shell prompt and updates the value of a specified variable by multiplying the variable's current value by a specified amount. The parameter <code>value</code> is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.
-------------	--

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example the current value of the variable `users` is 5. To change the value of this variable by multiplying by 10, use the following `gv_mul` command. Then, enter the `gv_read` command to get the changed value:

SEE ALSO Gv_mul, Gv_mulv

EMPOWER/CS-V1.0.1

SEE ALSO [gv_mul](#)

COMMAND gv_or

SYNTAX gv_or name value

DESCRIPTION	The <code>gv_or</code> command is entered at the command line and updates the value of a specified variable by applying a bit-wise OR masking operation to the variable. The parameter <code>value</code> is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.
-------------	---

This command operates on all variable types except strings.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

SEE ALSO Gv_or

FUNCTION	Gv_or, Gv_orv
SYNTAX	<pre>int Gv_or(name, value) char *name; int value; int Gv_orv(name, value, oldvalue) char *name;</pre>
DESCRIPTION	<p><i>Parameters</i></p> <p>name The name of the global variable</p> <p>value The operand for the operation</p> <p>oldvalue The pointer location where the original value should be stored (for Gv_orv())</p> <p><i>Comments</i></p> <p>The Gv_or() and Gv_orv() functions update the value of a specified variable by applying a bit-wise OR masking operation to the variable. These functions can be inserted in your script when editing the script file.</p> <p>Gv_or() is used if the variable is an integer, and Gv_orv() is used if the variable is not an integer.</p>
RETURN VALUE	Gv_or() returns the original value of the specified variable and Gv_orv() copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.
SEE ALSO	gv_or

COMMAND	<code>gv_parallel</code>
SYNTAX	<code>gv_parallel name</code>
DESCRIPTION	The <code>gv_parallel</code> command is entered at the shell prompt and waits for the value of the specified parallel variable to become greater than zero then decrements the variable by one. When used in conjunction with the <code>gv_unparallel</code> command, the parallel variable becomes an "on/off" switch to control multiple script execution.
RETURN CODE	If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.
SEE ALSO	<code>Gv_getparallel</code> , <code>Gv_parallel</code> , <code>Gv_setparallel</code> , <code>Gv_unparallel</code> , <code>gv_getparallel</code> , <code>gv_setparallel</code> , <code>gv_unparallel</code>

EMPOWER/CS-V1.0.1

The first 50 users to reach the `Gv_parallel()` function during script execution immediately will execute the limited portion of transactions. The remaining 450 users will reach the `Gv_parallel()` function and will wait.

EMPOWER/CS-V1.0.1

Gv_getparallel, Gv_setparallel, Gv_unparallel, gv_getparallel,
gv_parallel, gv_setparallel, gv_unparallel

EMPOWER/CS-V1.0.1


```

$ gv_init synch int 0
$ gv_protect synch
$ gv_stat

```

Name	Type	Value	Allocated	Protector
synch	int	0	0	CONSOLE

```

$ s1 &
[1] 3058
s1 ready
$ s2 &
[2] 3060
$ s3 &
[3] 3062
s2 ready
s3 ready
$ gv_stat

```

Name	Type	Value	Allocated	Protector
synch	int	0	3	CONSOLE

```

$ gv_unprotect synch
$ gv_stat

```

Name	Type	Value	Allocated	Protector
synch	int	3	3	NONE

SEE ALSO

Gv_protect, Gv_unprotect, gv_unprotect


```
Gv_alloc("users", "int");
Gv_protect("users");
if (Gv_test("users", "==", 3))
    Gv_write("users", 0);
Gv_unprotect("users");
Gv_inc("users");
```


Introduction

If the value of `balance` is not an integer, use the `Gv_readv()` function as in the following script file entries:

In the above example, the `Gv_readv()` function retrieves the current value of the variable `balance` and stores it in the floating point variable `curbalance`.

gv_read

COMMAND	gv_rm
SYNTAX	gv_rm [-f] [-r { name1 name2 ...}]
DESCRIPTION	The gv_rm command is entered at the shell prompt and removes the specified variables from shared memory. If a variable currently is allocated to a script, the command will fail. If the -f option is used, each variable will be removed even if it is allocated to a script. If the -f option is used to remove a variable allocated to a script, a warning message will appear on screen and the variable will be removed. The script will exit the next time it attempts to use the variable. The -r option removes all global variables.
RETURN CODE	If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.
EXAMPLES	The following example command will remove the variables users and transactions : \$ gv_rm users transactions
SEE ALSO	gv_seg

SEE ALSO [Gv_rshift](#), [Gv_lshift](#), [gv_lshift](#)

SEE ALSO Gv_lshift, Gv_lshiftrv

COMMAND	gv_seg
SYNTAX	gv_seg [number-of-variables -r]
DESCRIPTION	<p>The gv_seg command is entered at the shell prompt and creates a shared memory segment to support 128 global variables by default. When creating a new shared memory segment, the number-of-variables argument defines the number of variables the new segment will support. The -r option removes the existing shared memory segment which removes all existing global variables. If you want to create a new shared memory segment with a different size, the existing shared memory segment must be removed first.</p>
RETURN CODE	<p>If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.</p>
EXAMPLES	<p>To create a shared memory segment that supports 150 global variables, use the following command (assuming a shared memory segment does not already exist):</p> <pre>\$ gv_seg 150</pre> <p>If a shared memory segment already exists, as would be the case if a Global Variables Command had already been executed, the existing shared memory segment must be removed before the new segment is created:</p> <pre>\$ gv_seg -r \$ gv_seg 150</pre>
SEE ALSO	gv_init , gv_rm

SEE ALSO [Gv_getparallel](#), [Gv_parallel](#), [Gv_setparallel](#), [Gv_unparallel](#), [gv_getparallel](#), [gv_parallel](#), [gv_unparallel](#)

FUNCTION **Gv_setparallel**

```
SYNTAX      void Gv_setparallel(name, value)
            char *name;
            unsigned short int value;
```

DESCRIPTION	Parameters
-------------	------------

name	The name of the parallel global variable
------	--

value	The new value of the specified parallel variable
-------	--

Comments

The `Gv_setparallel()` function assigns a new value to the specified parallel variable. The new value is listed as the second parameter. You can insert this function into your script when editing your script file.

RETURN CODE	If the function succeeds, the return code is set to zero. If an error occurs, the script exits and an error message is sent to the standard error destination.
-------------	--

SEE ALSO [Gv_getparallel](#), [Gv_parallel](#), [Gv_unparallel](#), [gv_getparallel](#), [gv_parallel](#), [gv_setparallel](#), [gv_unparallel](#)

COMMAND **gv_stat**

SYNTAX **gv_stat** [name | -s]

DESCRIPTION The **gv_stat** command is entered at the shell prompt and returns status information to the standard output destination for all variables or for a specified variable. The parameter is either a variable name or "-s". If the parameter provides the name of a variable, status information is provided for that variable only. If the -s option is specified, summary information is provided indicating the number of global variables that exist compared to the number of variables possible.

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES The **gv_stat** command sends to the standard output destination a table showing the name, type, and value of the specified variable, with the number of scripts that have allocated the variable and the identity of the variable's protector, if applicable. For example:

\$ gv_stat users				
Name	Type	Value	Allocated	Protector

users	int	3	0	NONE

If **gv_stat** is executed without an argument, information will be listed for all variables:

\$ gv_stat				
Name	Type	Value	Allocated	Protector

users	int	3	0	NONE
a	int	-1	0	NONE
b	int	100	0	NONE


```
$ gv_stat -s
10/20 global variables exist
```

Gv_stat

DESCRIPTION	Parameters
	<p>name The name of the global variable</p> <p>info The structure where status information for the specified global variable is stored</p>

The `Gv_stat()` function copies the following global variable information into a structure specified in the second parameter: current value, name, type, owner name, owner process ID, and the number of scripts allocated to the variable. You can insert this function into your script when editing the script file.

The `gv_info` structure is declared in the `empowerGV.h` file and is included automatically in your script by the `EMPOWER/CS` tool `Csc`. This structure is demonstrated below:

```

struct gv_info {
    int index; /* >= 0 indicates allocated to this script */
    int owner; /* >= 0 indicates pid of protecting owner */
    int users; /* number of users allocated */
    char name[SHMAXNAMELEN];
    char type[SHMAXTYPELEN];
    union val {
        int i;
        char c;
        .... .;
    } value; /* current variable value */
};

```


If the specified variable is protected by another script, the `gv_info->value` field will be empty. The `gv_info->owner` field will contain the process ID of the protecting script.

EXAMPLES The following example demonstrates using `Gv_stat()` in a script:

```

struct gv_info info;

if( Gv_stat("users", &info) ) {
    if ( strcmp("int", info.type) ) {
        fprintf(stderr, "wrong type for global variable
users: %s\n", info.type);
        Exit(-1);
    }
}
else
    Gv_alloc("users", "int", 0);

Beginscenario("manager");
...
Endscenario("manager");

```

SEE ALSO [gv_stat](#)

DESCRIPTION	The <code>gv_sub</code> command is entered at the shell prompt and updates the value of a specified variable by subtracting the specified amount from the variable's current value. The parameter <code>value</code> is the operand for the operation. When this command is entered, the original value is written to the standard output destination before the new value, based on operation results, is assigned.
-------------	--

RETURN CODE If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.

EXAMPLES In the following example, each of the GV commands returns the current value of the variable to the standard output destination before performing the specified operation. Suppose the variable `users` has a current value of 5 and you wish to make the value 6, then change the value by subtracting 4. The interaction would be as follows:

```
$ gv_write users 6
5
$ gv_sub users 4
5
$ gv_read users
2
```

SEE ALSO Gv_sub

[illegible]

Gv_test


```
Gv_alloc("quitflag", "int");
if (Gv_test("quitflag", "==", 1))
    Exit(1);
```


COMMAND	gv_unparallel
SYNTAX	gv_unparallel name
DESCRIPTION	The gv_unparallel command is entered at the shell prompt and increments the value of the variable by one. When used in conjunction with gv_parallel , the parallel variable becomes an "on/off" switch to control multiple script execution.
RETURN CODE	If the command succeeds, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.
SEE ALSO	Gv_getparallel , Gv_parallel , Gv_setparallel , Gv_unparallel , gv_getparallel , gv_parallel , gv_setparallel

Introduction

SEE ALSO [Gv_protect](#), [Gv_unprotect](#), [gv_protect](#)


```
Gv_alloc("users", "int");
Gv_protect("users");
if (Gv_test("users", "==", 3))
    Gv_write("users", 0);
Gv_unprotect("users");
Gv_inc("users");
```


09697984-102610
09697984-102610

COMMAND gv_waituntil

SYNTAX gv_waituntil name relation-string [value]

DESCRIPTION	<p>The <code>gv_waituntil</code> command is entered at the shell prompt and compares the current value of the specified variable to the specified parameter value according to the test relation given in the parameter <code>relation-string</code>. Operations executing the command from the UNIX script driver or shell script will pause until the relational comparison is true. If the relation string is a logical comparison ("<code>\$</code>" or "<code>!</code>"), the <code>value</code> argument will be ignored.</p>
-------------	---

RETURN CODE	When the relational comparison becomes true, the return code is set to zero. If an error occurs, the return code is set to one and an error message is sent to the standard error destination.
-------------	--

SEE ALSO [Gv_waituntil](#), [Gv_waitwhile](#), [gv_waitwhile](#)

Cheryl Lin


```
Gv_alloc("users", "int");
Gv_inc("users");
Gv_waituntil("users", "==", 128);
```

SEE ALSO [Gv_waitwhile](#), [gv_waituntil](#), [gv_waitwhile](#)

2025

SEE ALSO [Gv_waitwhile](#), [Gv_waituntil](#), [gv_waituntil](#)

EMPOWER/CS-V1.0.1

SEE ALSO [Gv_waituntil](#), [gv_waituntil](#), [gv_waitwhile](#)


```
$ gv_write users 6
5
```

```
$ gv_read users
6
```

```
$ gv_write solution "world peace"
```

EMPOWER/CS-V1.0.1

EMPOWER/CS-V1.0.1

To assign a new value to the variable with the name `count`, use the following example if `count` is an integer:

To assign a new value to the non-integer variable `balance`, use:

SEE ALSO

gv_write

SEE ALSO Gv_xor


```
SYNTAX      int Gv_xor(name, value)
            char *name;
            int value;

            int Gv_xorv(name, value, oldvalue)
            char *name;
```

Comments

The `Gv_xor()` and `Gv_xorv()` functions update the value of a specified variable by applying a bit-wise EXCLUSIVE-OR masking to a variable. You can insert these functions into your script when editing your script file.

`Gv_xor()` is used if the variable is an integer, and `Gv_xorv()` is used if the variable is not an integer. The parameter `value` is the operand for the operation, and the parameter `oldvalue` (for `Gv_xorv()`) specifies the pointer location where the original value should be stored.

`Gv_xor()` returns the original value of the specified variable and `Gv_xorv()` copies the original value to a pointer location. After the value of the variable has been updated, the original value, cast as an integer, is returned. If an error occurs, the script exits and an error message is sent to the standard error destination.

gv_xor

2025 1000

Introduction

During script execution in Display mode, `InitialWindow()` attempts to locate the Windows listed as its parameters and place them in the same positions as captured. The function does not apply to Non-Display mode script execution.

Note: You should not attempt to remove or edit this function in your script file because you change the state of the windows desktop as it was captured, and, therefore run the risk of breaking the script.

EXAMPLES The following example demonstrates `InitialWindow()` in a script file:

The following example demonstrates the log file of an executed script that encountered an error because the desktop was not restored to its captured state:


```
>>> InitialWindow(2,"File Manager",0,0,MAXWIDTH,MAXHEIGHT)
Warning: Unable to find window
```

SEE ALSO

CurrentWindow

EMPOWER/CS-V1.0.1

2025

EMPOWER/CS-V1.0.1

The following example demonstrates using `Inote()` in a script to identify the number of a loop as it executes:

```
for (i=1; i<=10; i++){
    Inote(i); /* identify loop number */
    ...
}
```

SEE ALSO

Note

EMPOWER/CS-V1.0.1


```
Type(" ^I^I");
```

```
KeyPress (VK_ESCAPE);
```

KeyPress, KeyUp, SysKeyDown, SysKeyPress, SysKeyUp

Introduction


```
WindowRcv("Pt");  
KeyPress(VK_LEFT);  
KeyDown(VK_SHIFT);  
KeyPress(VK_TAB);  
KeyUp(VK_SHIFT);  
KeyDown(VK_CONTROL);  
  
Type("^I^I^I");  
  
KeyUp(VK_CONTROL);  
  
KeyPress(VK_ESCAPE);
```

SEE ALSO

KeyDown, KeyUp, SysKeyDown, SysKeyPress, SysKeyUp

EMPOWER/CS-V1.0.1


```
WindowRcv ("Pt");
KeyPress (VK_LEFT);
KeyDown (VK_SHIFT);
KeyPress (VK_TAB);
KeyUp (VK_SHIFT);
KeyDown (VK_CONTROL);

Type ("^I^I^I");

KeyUp (VK_CONTROL);

KeyPress (VK_ESCAPE);
```

SEE ALSO

KeyDown, KeyPress, SysKeyDown, SysKeyPress, SysKeyUp


```
AppWait(5.21);
WindowRcv("Pt");

LeftButtonDown(213,111);
```

(continued on following page . . .)

Sometimes, a comment is inserted by EMPOWER/CS before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

SEE ALSO

LeftButtonPress, LeftButtonUp, LeftDbIPress

EMPOWER/CS-V1.0.1


```
AppWait(5.21);  
WindowRcv("Pt");  
  
LeftButtonDown(213,111);  
  
AppWait(0.55);  
WindowRcv("Pt");  
  
LeftButtonUp(222,195);  
  
AppWait(0.38);  
WindowRcv("Pt");  
  
LeftButtonDown(340,216);  
LeftButtonUp(333,219);  
  
AppWait(0.17);  
WindowRcv("Pt");  
  
LeftButtonPress(338,220);
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."


```
/* Clicked (Button) (FETCH) */
LeftDblPress(276,345);
```

SEE ALSO [LeftButtonDown](#), [LeftButtonUp](#), [LeftDblPress](#)

FUNCTION	LeftButtonUp
SYNTAX	<pre>void LeftButtonUp(x,y) unsigned int x,y;</pre>
DESCRIPTION	<p><i>Parameters</i></p> <p>x The on screen x coordinate of the PC mouse</p> <p>y The on screen y coordinate of the PC mouse</p> <p><i>Comments</i></p> <p>The <code>LeftButtonUp()</code> function is inserted in the script file automatically during Capture when the user releases the left mouse button. During script execution in Display mode, this function is used to emulate releasing the left mouse button. In Non-Display mode, this function is used to emulate the pointer rate delay based on the <code>x,y</code> parameters.</p> <p>The <code>x,y</code> parameters indicate the position of the mouse on screen at the time the button was released during Capture.</p> <p>The <code>LeftButtonUp()</code> function may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.</p>
RETURN VALUE	(not applicable)
EXAMPLES	The following example demonstrates various left button activities:


```
AppWait(5.21);  
WindowRcv("Pt");  
  
LeftButtonDown(213,111);  
  
AppWait(0.55);  
WindowRcv("Pt");  
  
LeftButtonUp(222,195);  
  
AppWait(0.38);  
WindowRcv("Pt");  
  
LeftButtonDown(340,216);  
LeftButtonUp(333,219);  
  
AppWait(0.17);  
WindowRcv("Pt");  
  
LeftButtonPress(338,220);
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */  
LeftDbIPress(276,345);
```

SEE ALSO LeftButtonDown, LeftButtonPress, LeftDbIPress


```
AppWait(5.21);
WindowRcv("Pt");

LeftDblPress(213,111);

AppWait(0.55);
WindowRcv("Pt");
```


Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

SEE ALSO [LeftButtonDown](#), [LeftButtonPress](#), [LeftButtonUp](#)

FUNCTION	Log
SYNTAX	<code>void Log(str)</code> <code>char *str;</code>
DESCRIPTION	<p><i>Parameters</i></p> <p><code>str</code> A null-terminated string recorded into the log file of an executed script</p> <p><i>Comments</i></p> <p>You can insert the <code>Log()</code> function into your script when editing the script file. This function allows you to place additional information in a log, such as the value of a variable.</p> <p><code>Log()</code> records the parameter <code>str</code> in the log file. The parameter <code>str</code> is a null-terminated string and is recorded in the log file in the following format.</p> <pre>>>> 16 Log("str")</pre> <p><i>Note:</i> The number following the ">>>" is the line number corresponding to a line in the script file.</p>
RETURN VALUE	(not applicable)
SEE ALSO	Inote, Note

Introduction

Close, Closeenv, Logon

Comments

The database environment for the specified dbnum must have been opened with `Openenv()` before a `Logon()` call will execute. The `Logon()` function provides access to the database so that a cursor structure can be opened to process the SQL statement. Therefore, `Logon()` will occur after `Openenv()` and before `Open()` in the script. The specifications for `Username()` and `Password()` also must be called in the script before a `Logon()` function will successfully execute.

Note: Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured in the script file, you may drastically alter the expected behavior of the application and SUT and therefore, break your script during execution.

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

EXAMPLES The following script segment demonstrates the process of logging on to the database, ORACLE, from the logon structure LOG1:

```
Type("scott^Itiger^ITRAMP^M")

Username(LOG1, "scott@TRAMP");
Password(LOG1, "tiger");
Logon(ORACLE, LOG1);

AppWait(0.05);

Think(2.35);

...

Open(LOG1,CUR1);

Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,
SAL, COMM, DEPTNO FROM EMP, UPDATE OF EMPNO, ENAME,
JOB, MGR, HIREDATE, SAL, COMM, DEPTNO");
```

SEE ALSO Logoff, Open, Openenv, Password, Username

[illegible]

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

SEE ALSO

MouseButtonPress, MouseButtonUp, MouseButtonDoubleClick

0903-7681

RETURN VALUE (not applicable)

EXAMPLES The following example demonstrates various button activities:

```
AppWait(5.21);  
WindowRcv("Pt");  
  
LeftButtonDown(213,111);  
  
AppWait(0.55);  
WindowRcv("Pt");  
  
LeftButtonUp(222,195);  
  
AppWait(0.38);  
WindowRcv("Pt");  
  
MiddleButtonDown(340,216);  
MiddleButtonUp(333,219);  
MiddleButtonPress(338,221);  
  
AppWait(0.17);  
WindowRcv("Pt");
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */  
MiddleDbIPress(276,345);
```

SEE ALSO MiddleButtonDown, MiddleButtonUp, MiddleDbIPress

EMPOWER/CS-V1.0.1

EXAMPLES

The following example demonstrates various button activities:

```
AppWait(5.21);  
WindowRcv("Pt");  
  
LeftButtonDown(213,111);  
  
AppWait(0.55);  
WindowRcv("Pt");  
  
LeftButtonUp(222,195);  
  
AppWait(0.38);  
WindowRcv("Pt");  
  
MiddleButtonDown(340,216);  
MiddleButtonUp(333,219);  
MiddleButtonPress(338,221);  
  
AppWait(0.17);  
WindowRcv("Pt");
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */  
MiddleDbPress(276,345);
```

SEE ALSO

MiddleButtonDown, MiddleButtonPress, MiddleDbPress

2025 年 1 月 1 日

EXAMPLES

The following example demonstrates various button activities:

```
AppWait(5.21);  
WindowRcv("Pt");  
  
LeftButtonDown(213,111);  
  
AppWait(0.55);  
WindowRcv("Pt");  
  
LeftButtonUp(222,195);  
  
AppWait(0.38);  
WindowRcv("Pt");  
  
MiddleButtonDown(340,216);  
MiddleButtonPress(333,219);  
MiddleButtonUp(338,221);  
  
AppWait(0.17);  
WindowRcv("Pt");
```

Sometimes, a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */  
MiddleDbPress(276,345);
```

SEE ALSO

MiddleButtonDown, MiddleButtonPress, MiddleButtonUp

FUNCTION	Note
SYNTAX	<code>Note(str)</code> <code>unsigned char *str;</code>
DESCRIPTION	<p><i>Parameters</i></p> <p><code>str</code> The text of a message to be displayed in Monitor</p> <p><i>Comments</i></p> <p>You must insert the <code>Note()</code> function into the script when editing. The <code>Note()</code> function specifies a message that will appear in the "Note" column of Monitor View 5. The <code>str</code> parameter is the text of the message and this message must be contained within double quotes. When a script executes, all <code>Note()</code> statement will be listed in the log file.</p>
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.
SEE ALSO	Inote, Log

EMPOWER/CS-V1.0.1

The following example script segment demonstrates opening a cursor, CUR1, from the logon structure LOG1:

```
Logon(ORACLE1, LOG1);

AppWait(0.05);

Think(2.35);

...

Open(LOG1,CUR1);

Parse(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,
SAL, COMM, DEPTNO FROM EMP, UPDATE OF EMPNO, ENAME,
JOB, MGR, HIREDATE, SAL, COMM, DEPTNO");
Define(CUR1, "1", STRING, 5);
Define(CUR1, "2", STRING, 11);
Define(CUR1, "3", STRING, 10);
Define(CUR1, "4", STRING, 5);
Define(CUR1, "5", STRING, 10);
Define(CUR1, "6", STRING, 9);
Define(CUR1, "7", STRING, 9);
Define(CUR1, "8", STRING, 3);
Exec(CUR1);

Fetch(CUR1);
GetNextRow(CUR1);
```

Close, Logon, Openenv

EMPOWER/CS-V1.0.1

The following example demonstrates how `Openenv()` may appear in a script file. In this example, a Version V6V7 Oracle database was specified:

SEE ALSO

Closenv, Logon, Open

Comments

`Paceconstant()` causes transactions to be submitted at a constant throughput. It accepts an argument naming the pace and defining the speed of the pace. The second argument to `Paceconstant()` defines the number of seconds that the script should take since the last call to `Paceconstant()`. The first call to a pacing function does not delay; it is used as a starting point for the subsequent calls. For this reason, the first call to a pacing function is often made with arguments of 0 to define the speed.

EMPOWER/CS-V1.0.1

SEE ALSO Pacetne, Paceuniform

FUNCTION	Pacetne								
SYNTAX	<pre>Pacetne(str, min, ave, max) char *str; int min, ave, max;</pre>								
DESCRIPTION	<p><i>Parameters</i></p> <table><tr><td>str</td><td>An argument naming the pace and defining the speed of the pace</td></tr><tr><td>min</td><td>An integer specifying minimum delay for the speed of the pace</td></tr><tr><td>ave</td><td>An integer specifying the average delay for the speed of the pace</td></tr><tr><td>max</td><td>An integer specifying maximum delay for the speed of the pace</td></tr></table>	str	An argument naming the pace and defining the speed of the pace	min	An integer specifying minimum delay for the speed of the pace	ave	An integer specifying the average delay for the speed of the pace	max	An integer specifying maximum delay for the speed of the pace
str	An argument naming the pace and defining the speed of the pace								
min	An integer specifying minimum delay for the speed of the pace								
ave	An integer specifying the average delay for the speed of the pace								
max	An integer specifying maximum delay for the speed of the pace								

Comments

Pacing functions can be inserted into your script file to control the pace of the script. These functions cause the script to pause long enough to make the script send transactions to the SUT at a predetermined throughput. Typically used in a loop, these functions may be nested to permit controlled throughput of transactions within a larger transaction.

`Pacetne()` has an average throughput that will be maintained, but submission of transactions occur at frequencies other than that defined by a constant distribution.

`Pacetne()` accepts three arguments to identify the speed of the pace - a minimum, average, and maximum delay. The average will be maintained during sustained execution of the script. Each time `Pacetne()` is executed, it will delay for a number of seconds since the last execution, where the number is taken from a truncated, negative exponential distribution defined by the three values. In a typical such distribution, the average is relatively close to the minimum. For

Note: You are permitted to nest pacing functions in a script. If you do so, we recommend setting the starting point for a pace explicitly, i.e. with the speed argument of 0. This will ensure that an inner loop executed at a fixed pace will begin execution immediately every time that the loop begins.

SEE ALSO Paceconstant, Paceuniform

EMPOWER/CS-V1.0.1

2025 年 1 月

EXAMPLES The following example demonstrates selecting the data for a query in a SQL statement for the cursor structure CUR1:

SEE ALSO Exec, Fetch

FUNCTION	Password
SYNTAX	<pre>Password(lognum, password) int lognum; char *password;</pre>
DESCRIPTION	<p><i>Parameters</i></p> <p>lognum An identifier of a logon communication structure</p> <p>password The user password used to access the database</p> <p><i>Comments</i></p> <p>To successfully access or logon to the database, a Username and Password must be entered. The Password() function is inserted into the script file when a user password is entered to logon to the database. During script execution, this function is used in the process of accessing the SUT. This function will occur in the script before a Logon() function.</p> <p><i>Note:</i> Because this function is inserted into your script based on how the client application interacts with the SUT, you should not attempt to edit this function or remove it from the script. If you change this function from when it was captured you may drastically alter the expected behavior of the client and SUT and therefore, break the script during execution.</p>
RETURN VALUE	If the function is successful, zero is returned. If an error occurs, -1 is returned.
EXAMPLES	<p>In the following example, the script will attempt to logon to the database ORACLE on the logon structure LOG1 with the password tiger:</p>


```
Username(LOG1, "scott@FOO");
Password(LOG1, "tiger");
Logon(ORACLE, LOG1);
```

AppName, Hostname, Language, Servername, Username


```
BeginTransaction(LOG1);
....
Pause(LOG1, TRANS2);

Continue(LOG1, TRANS1);
Commit(TRANS1);
```


SEE ALSO [BeginTransaction](#), [Continue](#)

FUNCTION	Pointerrate
SYNTAX	<code>double Pointerrate(n)</code> <code>double n;</code>
DESCRIPTION	<i>Parameters</i> n The mouse pointer speed measured in points per second

Comments

`Pointerrate()` sets the emulated mouse pointer speed for a script. Pointer speed is measured in points per second which is specified in the parameter `n`. The default pointer rate `Pointerrate(100)` which is listed at the top of each script created by EMPOWER/CS. You may change this default when editing the script file. Multiple `Pointerrate()` functions may be used in a script to specify, for example, different pointer rates when using different applications.

During script execution, each time that the emulated user is supposed to be moving the PC mouse, the script will pause a length of time defined as the number of points to be moved times the specified pointer rate. For example, if the emulated user must move the mouse 100 points and the pointer rate is specified as 50 points per second (`Pointerrate(50)`) then the script will pause two seconds during each mouse point from one location to the next specified location.

Care must be taken when inserting `Pointerrate()` functions into a script. The response time statistics for scenarios and functions are generated from the user's perspective. Therefore, they will take into account the time it takes for the emulated user to move the PC mouse. This is particularly important when you are comparing the performance of similar scripts. Any `Pointerrate()` functions inserted into one script must be inserted in the same locations in the other script(s) to provide a meaningful comparison.

error if you set rate at less than zero, script will abort and you will receive an error message.

RETURN VALUE This function returns the old pointer rate value.

EXAMPLES In the following example, the `Pointerrate()` is set at the beginning of the script at 150 points per second:

```
/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */

Typerate(5);           /* Typing delay in CPS */
Pointerrate(150);      /* Pointerrate in PPS */
Thinkuniform(1,2.5);   /* Think delay */
Seed(getpid());        /* Seed random number generator */
Timeout(300, CONTINUE); /* What to do if function takes too long */
Dberror(CONTINUE);     /* What to do on Database errors */
Unset(NOTIFY);         /* Don't display warnings. I'll use Mon to find them */
```

SEE ALSO Typerate

FUNCTION	Range
SYNTAX	<pre>int Range(min, max) int min, max;</pre>
DESCRIPTION	<p><i>Parameters</i></p> <p>min An integer specifying the minimum value of the range</p> <p>max An integer specifying the maximum value of the range</p> <p><i>Comments</i></p> <p>Range() often is used with the Sleep() function to perform random execution delays. This function is inserted into the script file during script editing.</p> <p>Range() returns a random number.</p>
RETURN VALUE	Range() returns a random integer number between or equal to min and max.
EXAMPLES	<p>The following script segment is an example of how to perform a random execution delay from 0 to 60 seconds.</p> <pre>Sleep(Range(0, 60));</pre>
SEE ALSO	Seed, Sleep

EMPOWER/CS-V1.0.1

Sometimes a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

SEE ALSO [RightButtonPress](#), [RightButtonUp](#), [RightDbIPress](#)

Introduction


```
AppWait(5.21);  
WindowRcv("Pt");  
  
LeftButtonDown(213,111);  
  
AppWait(0.55);  
WindowRcv("Pt");  
  
LeftButtonUp(222,195);  
  
AppWait(0.38);  
WindowRcv("Pt");  
  
RightButtonDown(340,216);  
RightButtonUp(333,219);  
  
AppWait(0.17);  
WindowRcv("Pt");  
  
RightButtonPress(325,226);
```

Sometimes a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */  
RightDblPress(276,345);
```

SEE ALSO

RightButtonDown, RightButtonUp, RightDblPress

EMPOWER/CS-V1.0.1

Sometimes a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

SEE ALSO [RightButtonDown](#), [RightButtonPress](#), [RightDblPress](#)


```
AppWait(5.21);  
WindowRcv("Pt");  
  
LeftButtonDown(213,111);  
  
AppWait(0.55);  
WindowRcv("Pt");  
  
LeftButtonUp(222,195);  
  
AppWait(0.38);  
WindowRcv("Pt");  
  
RightButtonDown(340,216);  
RightButtonUp(333,219);  
  
AppWait(0.17);  
WindowRcv("Pt");  
  
RightButtonPress(325,226);
```

Sometimes a comment is added before a button event to add context to the script. In this example, the comment indicates double clicking a button called "Fetch."

```
/* Clicked (Button) (FETCH) */  
RightDbPress(276,345);
```

SEE ALSO

RightButtonDown, RightButtonPress, RightButtonUp


```
Commit(CUR1);

/* insert data into database */
Parse(CUR1, "insert empno, ename, empjob into emp_table");

Bind(CUR1, "empno", INT, 4);
Bind(CUR1, "ename", STRING, 30);
Bind(CUR1, "empjob", STRING, 20);

/* 123 refers to empno, Smith -- ename, typist -- empjob */
Data(CUR1, "123|Smith|typist");

Exec(CUR1);

Rollback(CUR1);
```

SEE ALSO Commit

FUNCTION	Seed
SYNTAX	<pre>int Seed(n) int n;</pre>
DESCRIPTION	<p><i>Parameters</i></p> <p>n The value used to seed the random number generator</p> <p><i>Comments</i></p> <p>Seed() seeds the random number generator used by EMPOWER/CS that produces random think time values.</p> <p>If you wish to generate the same sequence of think times during repeated script executions, you should always seed the random number generator in the script with the same value. If you wish to generate different sequences of think times for each script in a multi-user emulation, you should seed each random number generator in each script with a different value.</p>
RETURN VALUE	Seed returns a value of n.
EXAMPLES	<p>The following example Seed() function seeds the random number generator from an argument specified on the command line used to execute the script:</p> <pre>Seed(atoi(argv[3]));</pre>
SEE ALSO	Thinkconstant, Thinktne, Thinkuniform, Range

EMPOWER/CS-V1.0.1

The following example demonstrates the `Servername()` function in a script:

```
Servername(LOG1, "DBSERVER");
Username(LOG1, "joe");
Password(LOG1, "xapq");
Logon(SYBASE, LOG1);
```

AppName, Hostname, Language, Password, Username,


```
SYNTAX      long Set(n)
            long n;
```

DESCRIPTION	Parameters
n	The EMPOWER/CS option turned on

set() turns on EMPOWER/CS options during script execution. Valid options are listed below:

<u>OPTION</u>	<u>DESCRIPTION</u>
LCMD	log miscellaneous commands
FLUSH	flush SUT responses to the log that are detected after a pattern match in a WindowRcv()
NOBUF	don't use buffered writes to the log file
NOTIFY	display a message when a timeout occurs, execution is suspended, or execution resumes
BELL	ring the bell twice when a timeout occurs
LOGGING	enable logging

Default options are LCMD and LOGGING.

RETURN VALUE `set()` returns a long value that contains previously set options.

EXAMPLES The following script segment causes flushing of the remaining buffer after the pattern is found and immediate recording of every response character in the log file.

```
Set(FLUSH);/* turn flush buffer on*/
Set(NOBUF);/* turn immediate recording on */
```

SEE ALSO Unset

EMPOWER/CS-V1.0.1

FUNCTION SetVar

```
SYNTAX      SetVar(curnum, var, value)
            int curnum;
            char *var, *value;
```

DESCRIPTION	<i>Parameters</i>
	<code>curnum</code> A cursor communication structure
	<code>var</code> The name of the variable
	<code>value</code> The new value to be assigned to the variable

Comments

The `SetVar()` function assigns a new value to the specified variable in the parameter `var`. The new value is assigned in the parameter value. You can insert this function into your script file when editing.

RETURN VALUE After the new value has been assigned to the specified variable, the original value of the variable is returned.

EXAMPLES The following example demonstrates using this function in a script:

```
char *empname, *empno;  
  
...  
  
Parse(CUR1, "insert ename, empno, empjob into  
employee_table");  
  
Bind(CUR1, "ename", STRING, 50);  
Bind(CUR1, "empno", INT, 4);  
Bind(CUR1, "empjob", STRING, 30);
```

(continued on following page...)

SEE ALSO

EMPOWER/CS-V1.0.1

FUNCTION	Sleep
SYNTAX	<code>double Sleep(n)</code> <code>double n;</code>
DESCRIPTION	<i>Parameters</i> n The number of seconds to suspend script execution <i>Comments</i> Sleep() suspends script execution for n seconds simulating a think delay.
RETURN VALUE	Sleep() returns n.
EXAMPLES	In the following example, Sleep() specifies that script execution will suspend for 30 seconds: <code>Sleep(30);</code> You can also specify a range for the Sleep() parameter: <code>Sleep(Range(40,60));</code>
SEE ALSO	Think, Thinkconstant, Thinktne, Thinkuniform

Comments

During script execution, `Sql()` sends the SQL statement to the SUT and verifies that the syntax of the SQL statement is compatible with the SUT. The SQL statement is stored on the SUT to be executed with a subsequent `Exec()` call. This operation prepares the SUT for subsequent processing of the SQL statement with such functions as `Define()` or `Bind()`, `Describe()`, `Data()`, etc.

Note: You may edit this function to accept variable arguments in a way similar to the C function `printf()`. The `sql()` function will accept arguments so that data can be passed into the script from the command line. Arguments are passed to this function via the script execution statement. In the script execution command, arguments follow the names of the executable script file and the log file, and all arguments

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

```
Sql(CUR1, "SELECT EMPNO, ENAME, JOB, MGR, HIREDATE,  
SAL, COMM, DEPTNO FROM EMP, UPDATE OF EMPNO, ENAME,  
JOB, MGR, HIREDATE, SAL, COMM, DEPTNO");
```

SEE ALSO Exec, Fetch, Parse

BOOK REVIEW

SEE ALSO Exec, Parse, Sql

Exec, Parse, Sql

FUNCTION **Suspend**

SYNTAX `Suspend()` ;

DESCRIPTION `Suspend()` is used during script execution to suspend a script until the signal `SIGRESUME` arrives. Suspending script execution is useful when multiple scripts are executed simultaneously. Use of the suspend feature permits control and synchronization of concurrent scripts.

If the `LCMD` (a default option) option is set with `Set()`, a suspended message with the time stamp of the suspension will be recorded in the log file. A sample of the suspend message in the log file is shown below:

```
>>>     33 Suspend() 13:37:06.96
```

Once a script has been suspended, execution can be resumed in three ways. First, if the script was started with the `Mix` tool, the execution of the script can be resumed with the `Mix resume` command. Second, if the script is started at the UNIX script driver shell prompt, the UNIX `kill` command can be used to send the `SIGRESUME` signal to the script process. See `$EMPOWER/h/empower.h` for the `SIGRESUME` value on your system. You also can resume scripts from Monitor with the "R" commands

If the `LCMD` (a default option) option is set, a resume message with a time stamp will be recorded in the log file. A sample of the resume message is shown below:

```
>>>     33 Resume() 13:37:30.54
```

RETURN VALUE If the function is successful, zero is returned. If an error occurs, -1 is returned.

SEE ALSO `kill(1)` in your UNIX User's Guide

FUNCTION **SysKeyDown**

SYNTAX `void SysKeyDown(key)`
 `unsigned int key;`

DESCRIPTION *Parameters*
 `key` A Microsoft Windows virtual key code value

The `SysKeyDown()` function is inserted in the script file automatically during Capture to indicate that a keyboard key on the PC was depressed while the Alt key was held down. 'Sys' implies that the keystrokes are being sent to the System Menu of the current window. During script execution in Display mode, this function is used to emulate pressing the specified key down with the Alt key. In Non-Display mode, this function emulates type delay for pressing the key.

The parameter `key` represents a Microsoft Windows virtual key code value. Some examples of valid virtual key codes are `VK_SPACE`, `VK_DELETE`, `VK_BACK`, `VK_DOWN`, and `VK_UP`.

Key events may be edited in your script file, but because you change the activity as it was captured, you run the risk of breaking the script.

RETURN VALUE (not applicable)

EXAMPLES The following example script segment demonstrates using various key events. In this case, the user pressed the Alt key (`VK_MENU`) and then pressed and released the F key. Then the user released the Alt key and pressed the left arrow key, and pressed and released the c key:


```
CurrentWindow("Program Manager - [Empower/CS]",36,24,324,543);

Think(2.14);

SysKeyDown(VK_MENU);
SysKeyPress('F');
SysKeyUp(VK_MENU);
KeyPress(VK_LEFT);

Type("c");
```

SEE ALSO

KeyDown, KeyPress, KeyUp, SysKeyPress, SysKeyUp

EMPOWER/CS-V1.0.1


```
CurrentWindow("Program Manager - [Empower/CS]", 36, 24, 324, 543);

Think(2.14);

SysKeyDown(VK_MENU);
SysKeyPress('F');
SysKeyUp(VK_MENU);
KeyPress(VK_LEFT);

Type("c");
```

SEE ALSO [KeyDown](#), [KeyUp](#), [SysKeyDown](#), [SysKeyPress](#), [SysKeyUp](#)

[illegible]

SEE ALSO [KeyUp](#), [KeyPress](#), [KeyDown](#), [SysKeyDown](#), [SysKeyPress](#)

FUNCTION

```
double Think(n);
double n;
```

Parameters

n The amount of think delay measured in seconds

During script execution, `Think()` performs a delay to emulate a user's think time. The delay will be determined by the current think time distribution.

The think time distribution can be defined by any of the four think time distribution functions:

Thinkactual()	-	actual think time
Thinkconstant()	-	constant distribution
Thinktne()	-	truncated negative exponential distribution
Thinkuniform()	-	uniform distribution

`Think()` functions are inserted in the script file during Capture when the EMPOWER/CS user pauses before performing any activity. During Capture, you may specify the number of seconds that EMPOWER/CS will wait before inserting a `Think()` function into the script file. The `Think()` function is inserted in the captured script file only after the specified time has elapsed. The parameter of the `Think()` function specifies the amount of seconds elapsed.

Because the script was captured with a think time value, you must always have a value specified in the `Think()` functions during script execution (even if the value is zero) regardless of the think time distribution.

RETURN VALUE Think() returns the amount of delay time incurred.

```
Think(5.11);
ButtonPush("Employee Records|Next",726,439);

AppWait(2.09);
WindowRcv("SfPt");
ButtonPush("Employee Records|Next",726,439);
ButtonPush("Employee Records|Delete",714,344);

WindowRcv("SfPt");

Think(3.70);
ButtonPush("Employee Records|Close",714,157);
```

```
Thinkconstant(3);
Think(1.23);
```

EMPOWER/CS-V1.0.1

EMPOWER/CS-V1.0.1


```
/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */

Type(5);          /* Typing delay in CPS */
Pointerrate(150); /* Pointerrate in PPS */
Thinkactual();    /* Think delay */
Seed(getpid());   /* Seed random number generator */
Timeout(300, CONTINUE); /* What to do if function takes too long */
Dberror(CONTINUE); /* What to do on Database errors */
Unset(NOTIFY);    /* Don't display warnings. I'll use Mon to find them */

Beginscenario("example1");
```

SEE ALSO

Think, Thinkconstant, Thinktne, Thinkuniform


```
Thinkconstant(3);
Think(0);
```


10967


```
Thinktne(2.0, 5.0, 25.0);  
Think();
```

Seed, Think, Thinkactual, Thinkconstant, Thinkuniform

EMPOWER/CS-V1.0.1

EXAMPLES

The following script segment is an example of how to generate think times from a uniform distribution between 30 and 45 seconds and perform think delay only when `Think()` functions are executed.

```
Thinkuniform(30, 45);  
Think();
```

SEE ALSO

Seed, Think, Thinkactual, Thinktne, Thinkuniform


```
SYNTAX      void Time(p)
            struct timevalue *p;
```

Comments

```
struct timeval {
    long sec;    /* seconds since Jan 1. 1970 */
    short hsec; /* and hundredths of a second */
}
```

RETURN VALUE (not applicable)

SEE ALSO Eventtime, Difftime

FUNCTION	Timeout
SYNTAX	<pre>int Timeout (n, f) int n; int (*f) ();</pre>
DESCRIPTION	<p><i>Parameters</i></p> <p>n An integer that specifies the elapsed time (in seconds) after which a timeout should occur</p> <p>f The function to execute when a timeout occurs</p>

Comments

During script execution, `Timeout()` specifies if and when a timeout should occur during response reading from the server. The default function `Timeout(300, CONTINUE)` is inserted at the top of every EMPOWER/CS script created. You can change this default when editing your script file.

To execute a script successfully, all events on the server must occur in the same way as they occurred during Capture. If an event occurs other than the set of expected events, the script will continue to wait for the expected events to occur. A timeout identifies expected events that do not occur.

Occasionally, script execution can not continue because of unexpected events on the server or in Display mode the client or server. The resulting unexpected responses can identify an abnormal error, loss of data between the client and server, or an unexpected screen image. If your executing script encounters a timeout, a `WindowRcv()` may need to be modified. If a timeout occurs when an emulated user attempts to connect to the SUT, you should ensure that the database on the server is available.

The function (f parameter) designated to handle the timeout condition can be either an EMPOWER/CS function or a user-defined function.

If a timeout occurs during script execution, review the log file to determine the cause. In some cases, the expected behavior did occur, but it took longer than the current timeout delay. If this is the case, you should edit the script to increase the timeout threshold with a new `Timeout()` function.

RETURN VALUE	<code>Timeout()</code> returns the previous timeout value. An <code>ETIMEOUT</code> value will be returned by the executing function upon timeout.
EXAMPLES	The following is an example of how a <code>Timeout()</code> function may appear within a script: <pre>Timeout(60, EXIT);</pre>
SEE ALSO	<code>Dberror</code> , <code>Set</code> , <code>Unset</code>

EMPOWER/CS-V1.0.1

EMPOWER/CS-V1.0.1

RETURN VALUE This function returns the old typerate value

```
/* EMPOWER/CS V1.0.1 Remote Terminal Emulator Script */

Type(5);          /* Typing delay in CPS */
PointRate(150);   /* PointRate in PPS */
Think(1,2.5);     /* Think delay */
Seed(getpid());   /* Seed random number generator */
Timeout(300, CONTINUE); /* What to do if function takes too long */
Dberror(CONTINUE); /* What to do on Database errors */
Unset(NOTIFY);    /* Don't display warnings. I'll use Mon to find them */

BeginScenario("foo");
```

SEE ALSO Pointerrate

FUNCTION **Unset**

SYNTAX `long Unset (n)`
 `long n;`

DESCRIPTION *Parameters*
 n The EMPOWER/CS option to be turned off

Comments

`Unset ()` turns off EMPOWER/CS options during script execution.
Valid options are listed below:

<u>OPTION</u>	<u>DESCRIPTION</u>
LCMD	log miscellaneous commands
FLUSH	flush SUT responses to the log that are detected after a pattern match in a <code>WindowRcv()</code>
NOBUF	don't use buffered writes to the log file
NOTIFY	display a message when a timeout occurs, execution is suspended, or execution resumes
BELL	ring the bell twice when a timeout occurs
LOGGING	enable logging

Default options are LCMD and LOGGING.

RETURN VALUE `Unset ()` returns a long value containing the previously set options.

EXAMPLES The following examples demonstrate using `Unset ()`.

```
Unset (FLUSH);  
Unset (LCMD);
```

SEE ALSO **Set**


```
Type("scott^Itiger^IFOO^M")

Username(LOG1, "scott@FOO");
Password(LOG1, "tiger");
Logon(ORACLE1, LOG1);
```


EMPOWER/CS-V1.0.1

The following example demonstrates typical `WindowRcv()` commands from a script file:

```
AppWait(0.05);
WindowRcv("CwPtPtDwCoSfCwCwCwCwCwCwCwCwCwSfAcSzPt");
WindowRcv("PtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPtPt");
CurrentWindow("Run", 429, 491, 880, 764);
```

SEE ALSO

AppWait

All warning and error messages are listed in alphabetical order followed by detailed descriptions and corrective actions.

The input files to Draw do not contain the same unit on the line identifying the duration of the reports. Make sure your INPUT variable identifies the correct input files. Do not edit the .STD files created by Report.

An arithmetic operation (e.g., `Gv_add()`) was attempted on a character string type global variable. Only read and write operations are valid for character string variables.

Eventtime() was passed an invalid argument. Check script for mistyping. Make sure that you call Eventtime() with an argument that is one of TBF, TEF, TBS, or TES.

A `-w` argument to `Report` was followed by an invalid value. The value must specify a time in `ss`, `mm:ss`, `hh:mm:ss` format. The time can optionally be followed by a decimal point and an integer to identify a fraction of a second. Check the syntax of the `Report` command and rerun.

Draw was unable to separate the bars in a chart with space that is proportional to the number of users represented by each bar. Select a different set of input files with different numbers of users if you must have bars that are proportionally spaced.

A bit-wise operation (e.g., `Gv_and()`) was attempted on a double, float, or character string type global variable. Only read, write, and arithmetic operations are valid for

Can't continue. Specification generated is stored in
[specification]

Can't continue. Specifications generated are stored in [specification]

Draw encountered a condition that prevented it from continuing. Correct the condition. Specifications created prior to the error condition are saved in the file specified.

Can't create shared memory segment

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

Can't create global variable: maximum exceeded

An attempt was made to create an additional global variable beyond the limit allowed by default. If more global variables are required, remove all existing variables with the `gv_seg -r` command which will destroy all global variables, then, with `gv_seg newlimit` where `newlimit` is a new number, create a new shared segment with a higher limit. Use `gv_stat -s` to determine the current limit.

Can't create shared memory segment

Can't detach from global variable segment

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

Csc is trying to execute the C compiler on your UNIX script driver. The C compiler is not in your `PATH` environment variable or does not have execute permission. Ensure that the C compiler can be executed at the command line.

An input file to Draw does not contain a line identifying the number of users in the report. Make sure the input file is a .STD file. Do not edit the .STD files created by Report.

A Draw specification identifies an event that does not exist in all of the input .STD files. An event is the name of a scenario or function. The event must exist in all input files. Correct the specification to identify an existing event. Check spelling. Make sure your INPUT variable identifies the correct input files.

A Draw specification identifies an event that does not exist in the input .STD file. An event is the name of a scenario, function or transaction. Correct the specification to identify an existing event. Check spelling. Make sure your INPUT variable identifies the correct input file.

A Draw specification identifies a field that does not exist in the input `.STD` file. Check spelling in the specification. Make sure your `INPUT` variable identifies the correct input file.

A Draw specification identifies a field that does not exist in all of the input `.STD` files. Check spelling in the specification. Make sure your `INPUT` variable identifies the correct input files.

An input file to Draw is missing the line containing the duration of a report. Make sure your `INPUT` variable identifies the correct input files. Do not edit the `.STD` files created by Report.

CscC can not find a function library or header file required for compilation. Make sure your EMPOWER environment variable is set to the location of the installed EMPOWER/CS software. Make sure the EMPOWER environment variable is exported.

Make sure the UNIX script driver kernel is configured with adequate resources to execute processes. Check the UNIX script driver console for kernel error messages. Typical resources that need increasing are the maximum number of processes in a system and the maximum number of processes per user.

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs`

shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

Can't open `/dev/null`

An attempt to open `/dev/null` failed. Make sure `/dev/null` exists on your UNIX script driver and has read and write permission. Make sure `/dev/null` is a special character device.

Can't open current directory

The name of the current directory can not be obtained. A call to `getcwd()` failed. Make sure the current directory has read permission.

Can't open `[log]` for writing

Log specified can't be opened. Check syntax of command to execute your script. Check script table entries if you are executing scripts with Mix. Remember that a port must be specified to specify a log. Make sure the directory in which the log is being written has write permission for you. Make sure an existing copy of the log has write permission for you.

Can't open `[input]`

Can't open `[output]`

Can't open `[mixlog]`

Can't open `[csc*.o]`

Can't open `[script table]`

Can't open `[script.c]`

Can't open `[specification]`

An attempt to open a file failed. If the file is to be created, make sure you have write permission to the directory in which the file will be placed and write permission to an existing version of the file if the file exists. Make sure the file system in which the file is to be created is not full. If the file is to be read, make sure the file exists and that you have read permission on the file.

Csc++ can not read a function library or header file required for compilation. Make sure that you have read permission on the EMPOWER/CS libraries and header files.

EMPOWER/CS rebinds the standard input file descriptors of a script running in the background to prevent the script from receiving interrupt signals generated at the keyboard. A `dup()` function failed. The `dup()` was interrupted by a signal or reached a maximum number of open files. Make sure there is an adequate number of open files permitted on the UNIX script driver machine and that the maximum number of open files per user is large.

Can't remove allocated variable

Can't remove shared memory segment

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

Can't start [scriptid.n] .

Make sure the UNIX script driver kernel is configured with adequate resources to execute processes. Check the UNIX script driver console for kernel error messages. Typical resources that need increasing are the maximum number of processes in a system and the maximum number of processes per user.

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

A script executed the `Gv_unprotect()` function when the specified variable had not been protected by the script.

A script executed the `Gv_waitwhile()` or `Gv_waituntil()` function on a variable which the script has protected with `Gv_protect()`. A protected variable can not be updated by other scripts, so a `Gv_waitwhile()` or `Gv_waituntil()` function probably will cause an indefinite delay.

Draw encountered a keyword in a specification that was not followed by an = character. Check the specification.

Draw encountered a statement in a specification that did not contain a required character. Check the specification.

Character string in the specification is too long

The name of an event in an input file to Draw is too long. The maximum is 15 characters. Make sure your `INPUT` variable identifies the correct input files. Do not edit the `.STD` files created by Report.

Characters should be separated with spaces

The **LEGEND** characters in a specification to Draw are not separated by spaces. Check the syntax of the **LEGEND** statement in the specification.

COMMENT too long. Ignored

Draw requested a COMMENT in interactive mode and your response identified one that is too long. Enter a shorter COMMENT. Maximum COMMENT is 60 characters. Enter quit to exit Draw.

```
Current directory does not contain any ..STD files
```

Draw can not find any files with the `..STD` suffix in the current directory. Make sure you are in the directory containing your `.STD` files.

Division by zero undefined

The value zero was passed as the divisor argument to a `Gv_div()` or `Gv_mod()` function.

Empty line. Try again

Draw is requesting input in interactive mode and none was entered. Enter a response or type quit to exit Draw.

Error in [file]

A specification or input file to Draw contains an error. A specific error message follows this error. Refer to the description of the specific error message for more information.

Error near line [n] => [data]

A specification or input file to Draw contains an error. A specific error message follows the => symbol. Refer to the description of the specific error message for more information. The line number *n* is often not correct.

EVENT value is not defined properly

A specification to Draw contains an invalid **EVENT** statement. Check syntax in the specification.

Failed set all semctl

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the **ipcs** shell command. Terminate the scripts and restart them. Removing the shared memory segment with **ipcrm** which will destroy the global variables and recreating them with **gv_init** may be necessary.

Forced to use HIDDEN format

A Draw specification requested bars to be drawn in **CLUSTERED** format. Draw was unable to fit all of the bars on the chart. Draw changed format to **HIDDEN**. Reduce the number of bars to be drawn if you want the chart in **CLUSTERED** format.

Forcing protection of global variable protected by other

The **gv_protect** command was used with the **-f** option while the global variable was protected by a script.

The `gv_unprotect` command was used with the `-f` option while the global variable was protected by a script.

Draw requested input in interactive mode and your response was entered in an unacceptable format. Correct the format and retry. Enter a response or type quit to exit Draw.

A script attempted to allocate access to a variable which has already been allocated to the script. Each variable may be allocated only once in each script unless it has been de-allocated with `Gv_free()`.

A script attempted to protect a variable that is already protected. The variable must be unprotected before it can be protected again.

A specified variable was not created with the `gv_init` command, or was removed with the `gv_rm` command.

Generally, this error is caused when a `fork()` system call has been executed after a variable was allocated to the script. If this error occurs, you can not `fork()` after allocating variables.

Global variable not currently allocated

A Global Variable function specified a variable that is not currently allocated to the script. You must allocate a variable with the `Gv_alloc()` function before using the variable.

Global variable protected by other process

The `gv_protect` command was executed to protect a variable which is currently protected by a script.

Illegal global variable check type

An illegal check type was specified. Use only the commands and functions specified in this reference manual.

Illegal global variable relation

Illegal global variable relation length

Invalid global variable relation

An illegal relation string was used with the `Gv_test()`, `Gv_waitwhile()`, or `Gv_waituntil()` function.

Illegal return value from `semop/ctl` call

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command. Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` which will destroy the global variables and recreating them with `gv_init` may be necessary.

Incomplete specification -- required fields undefined

Draw encountered a specification that did not contain the required statements `INPUT`, `X`, `Y` and `EVENT`. Correct the specification.

INPUT file does not contain proper data
INPUT files do not contain proper data
One of the input files to Draw contains erroneous data. Make sure you specify
.STD files created by Report as input. Do not edit the .STD files that are to be used
by Draw.

INPUT was redefined; the last one will override
More than one INPUT statement was encountered in a specification to Draw. Draw
will use the last one found. Correct the specification.

Invalid character in name of scenario
A `Beginscenario()` or `Endscenario()` function contained unprintable characters.
Check the argument of the failing function. The argument must be a string.

A `Timeout()` function was called with an invalid argument. The argument must be an integer or floating point greater than or equal to zero. Check the argument of the failing function.

Invalid values in think time distribution

A `Thinkconstant()`, `Thinkuniform()`, or `Thinktne()` function was called with one or more invalid arguments. The arguments to one function must be ascending in value since they specify minimum, average and maximum values in sequence. Remember that `Thinkconstant()` requires one argument, `Thinkuniform()` requires two arguments, and `Thinktne()` requires three arguments. Check the arguments of the failing function.

LEGEND was redefined; the last one will override

More than one **LEGEND** statement was encountered in a specification to Draw. Draw will use the last one found. Correct the specification.

Missing quote

A line in the script table does not contain an even number of " characters. The " is used when specifying arguments to scripts in the script table. Check the syntax of your script table.

Missing script id

A line in the script table does not contain a script ID. Make sure you have identified the proper script table on the `Mix use` command. Check syntax of lines in the script table. Remember that comments in the script table begin with a `#` character.

Modulo operation invalid for type double

Modulo operation invalid for type float

Modulo arithmetic is not permitted for double or float type variables.

The `-m` option of `Report` must occur before the `-w` option. Check the syntax of the `Report` command and rerun.

Draw encountered an input file than contained the N/A symbol for a data value that you requested. The N/A is generated by Report if the event never occurred during the emulation. Rerun the emulation and make sure that the event requested occurs at least once or select a different event to chart.

A `Thinkconstant()`, `Thinkuniform()`, or `Thinktne()` function was called with one or more invalid arguments. The arguments must be integers or floating points greater than or equal to zero. Remember that `Thinkconstant()` requires one argument, `Thinkuniform()` requires two arguments, and `Thinktne()` requires three arguments. Check the arguments of the failing function.

A script was compiled with the `-h` option of Scc and subsequently executed with an invalid argument. The help information regarding the usage of arguments was not compiled in the script. Check the syntax of the command used to start the script or compile the script without the `-h` option to Scc and rerun.

The maximum number of comments in a specification to Draw was exceeded. The maximum number is 50. Reduce the number of comments.

Out of memory

The UNIX script driver machine is out of virtual memory.

Percentile out of range

A `-p` argument to Report was followed by an invalid value. The value must be an integer greater than zero and less than or equal to one hundred. Check the syntax of the Report command and rerun.

Performance measures in [input] not consistent

The input files to Draw do not contain the same set of Y values generated by Report. Make sure that you use the same set of `-p` and `-w` options to Report when creating each of the `.STD` files used as input to Draw. Make sure your `INPUT` variable identifies the correct input files.

Reached another BEGIN before keyword END

Draw encountered a BEGIN statement in a specification before an END statement terminated the current specification. Check the specification. Make sure you are identifying the correct specification.

Reached end of file before keyword END

Draw encountered the end of a specification before an END statement terminated the current specification. Check the specification. Make sure you are identifying the correct specification.

Received signal n

EMPOWER/CS received a signal that it was not expecting. The signal was probably generated by pressing the interrupt key, e.g. `^C`, a kill command or after detecting the disconnection of a terminal. Prevent the signal from recurring and rerun. You can locate the meaning of the signal in `/usr/include/sys/signal.h`.

Removing allocated variable

The `gv_rm` command was used with the `-f` option while the variable was allocated to at least one script.

Resumed at [99:99:99.99]

Execution of a script is continuing following a `Suspend()` function. The script was resumed from Mix, Monitor, or with a `kill` command.

Semctl call failed

Semop/ctl failed

Semop/ctl call failed

A problem occurred when attempting to access the UNIX script driver's shared memory segment. Check the access permissions of the segment with the `ipcs` shell command and increase the number of semaphore undo structures (`Sem undo` or `sem nu`--These structures are UNIX kernel parameters). Terminate the scripts and restart them. Removing the shared memory segment with `ipcrm` to destroy the global variables and then recreating them with `gv_init` may be necessary.

Source [script.c] newer than binary. execution continues.

A script detected that the source version of the script has a later modification date than the binary version. The source version may have been changed and requires a recompilation. Recompile the source version of the script or move the script to another directory to eliminate the warning.

The script looks in the current directory for the source version. The name of the source version is formed by adding a `.c` to the name of the binary.

Suspended at [99:99:99.99]

Execution of a script is suspended because of execution of the `Suspend()` function or Monitor. Script execution will pause until the script is resumed.

The timeout condition in a script was set to `CONTINUE`. A timeout occurred while the script was waiting for a response from the SUT. The expected response from the SUT was not received. You should examine the log created by the script and determine whether or not the response was valid.

The timeout condition in a script was set to `EXIT`. A timeout occurred while the script was waiting for a response from the SUT. The expected response from the SUT was not received. You should examine the log created by the script and determine whether or not the response was valid.

A TITLE in a specification to Draw is too long. The maximum is 60 characters. Reduce the number of characters in the TITLE.

Draw requested a YMIN and YMAX in interactive mode and your response did not specify both. Enter both values in integer or floating point format. Enter quit to exit Draw.

The maximum number of arguments that can be passed to a cc statement by Csc is 99. The limit has been exceeded. Decrease the number of arguments by modifying your E_CFLAGS and E_LIBS environment variables.

Draw encountered a statement in a specification that contained too many arguments after a = character. Check the syntax of the specification.

The maximum number of comments in a specification to Draw was exceeded. The maximum number is 50. Reduce the number of comments.

Too many fields. Only one field required

Draw requested input in interactive mode and your response identified too many choices. Enter only one choice from the list displayed by Draw. Enter `quit` to exit Draw.

Too many fields. Select up to 3 fields. Try again

Draw requested input in interactive mode and your response identified too many choices. Enter only up to three choices from the list displayed by Draw. Enter `quit` to exit Draw.

Too many files. Select up to 59 files. Try again

Draw requested input in interactive mode and your response identified too many choices. Enter only up to 59 choices from the list displayed by Draw. Enter `quit` to exit Draw.

Too many INPUT files -- extras were ignored

A specification to Draw identified too many INPUT files. The maximum number of input files is 59. Reduce the number in the specification.

Too many LEGEND characters -- extras were ignored

A specification to Draw identified too many LEGEND characters. There should be one LEGEND character for each Y value. Reduce the number in the specification.

Too many ORGANIZE values -- extras were ignored

A specification to Draw identified too many ORGANIZE values. There should be only one ORGANIZE value. Reduce the number in the specification.

Unexpected suspend request

A global variables shell command received a signal from a `kill` command or process. Someone else may be trying to kill your process.

Unit of time inconsistent. Try again

The input files selected Draw do not contain the same unit on the line identifying the duration of the reports. Select a different set of `INPUT` files from the list displayed by Draw. Do not edit the `.STD` files created by Report.

Unit of time inconsistent in [input]

The input files to Draw do not contain the same unit on the line identifying the duration of the reports. Make sure your `INPUT` variable identifies the correct input files. Do not edit the `.STD` files created by Report.

Unrecognized keyword -- line ignored

Draw encountered a statement in a specification that did not begin with a valid keyword. Check the specification. Make sure the keywords are followed by a space and a `=` character.

Value conversion failed

The value passed could not be converted to the variable type.

Value in the input file is missing -- N/A assumed

Draw encountered an input file that did not contain a `Y` value for an event that you requested. Rerun the emulation and make sure that the event requested occurs at least once or select a different event to chart. Do not edit the `.STD` files created by Report.

Within value out of range

X axis TITLE must contain 30 or fewer characters

X was redefined; the last one will override

Y axis TITLE must contain 30 or fewer characters

Y values are not consistent

Y was redefined; the last one will override

EMPOWER/CS-V1.0.1

[This page intentionally left blank]

09697994.102600

Contact PERFORMIX Technical Support if you experience problems with any aspect of EMPOWER/CS. For problems with script development, script execution, reporting, or other operational aspects of EMPOWER/CS, please have the following information at hand when you contact us:

- ☐ Any relevant script files
- ☐ Any relevant log files
- ☐ The Mix table
- ☐ The Mix log file
- ☐ Your software Version Number
- ☐ Your software Serial Number

To determine your software version and serial numbers, use any EMPOWER/CS tool to display the copyright banner. For example:

```
$ csc -
Csc:  EMPOWER/CS V1.0.1, Serial#R00000-000, Copyright PERFORMIX, Inc.
1988-95
Usage:
```

EMPOWER/CS software is identified with a three-digit version number, such as "1.0.1". The first digit represents the major version cycle, the second digit represents the minor version cycle, and the third digit represents the patch level. This patch level increases with bug fixes.

The serial number displayed in the copyright banner has two elements. The first portion identifies the five-digit software license number and is used by PERFORMIX support personnel to identify your particular platform, restrictions, and terms and conditions. The second part of the serial number identifies the copy number. The letter preceding the serial number identifies the type of license (i.e., "R" for regular, "S" for site, etc.).

When calling for technical support, you will need to give both the three digit version number and the entire serial number.

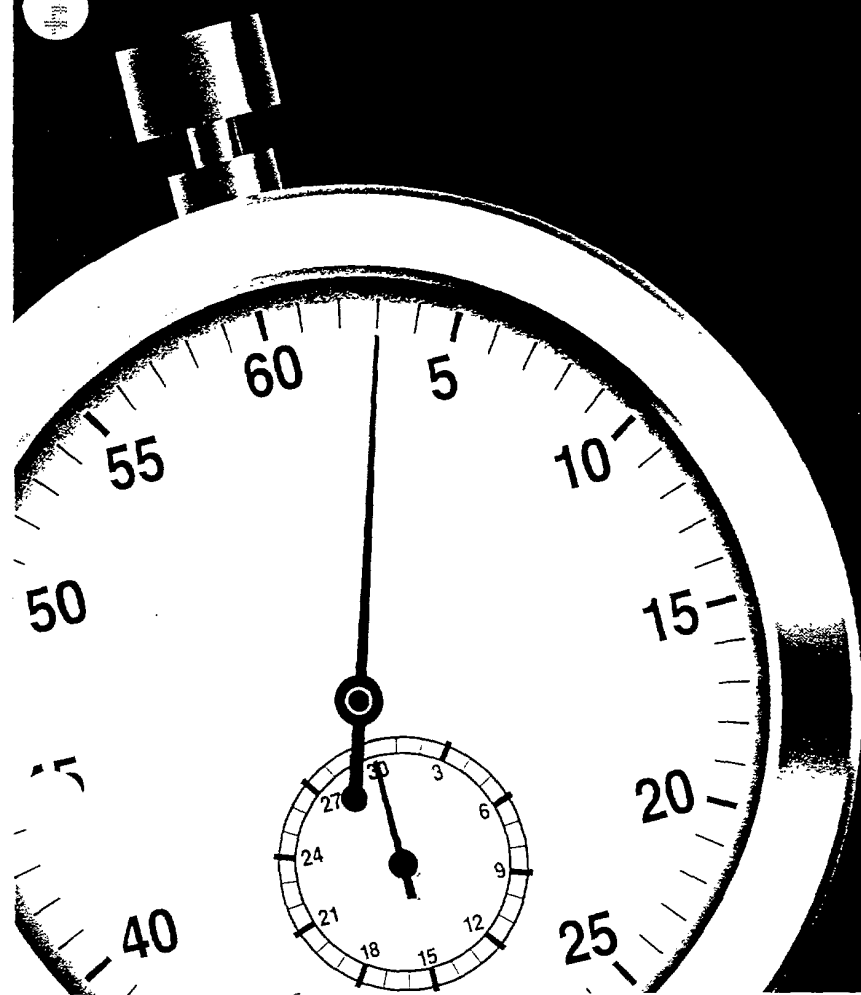
You also may email relevant information to support@performix.com.

EMPOWER

Load Testing Software

for

Client/Server
Applications

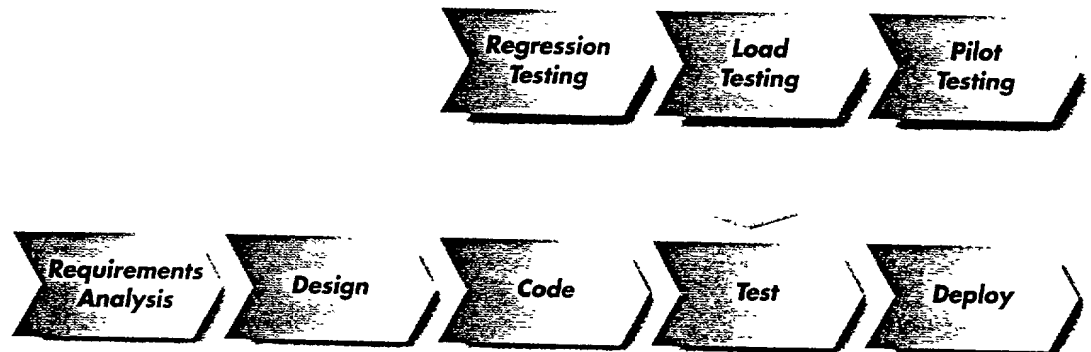


PERFORMIX

The testing process

Applications that are not load tested often fail during the pilot stage — and consequently **BUST** at full-scale deployment

The testing process is integral to successful application development. It consists of three stages—regression, load and pilot testing—of which load testing is the keystone.



Successful Application Development

The complete testing process is a triumvirate of regression, load and pilot testing. The process should, ideally, be completed in that order.

Regression testing (also called functional or quality testing) helps avoid errors linked to a single user. It is a linear process that examines virtually all pathways through an application. Since regression testing quickly isolates inconsistencies in the interface, it is particularly useful as applications are slightly changed. Sometimes a bit of tweaking in one area produces adverse affects in seemingly unrelated spots.

Load testing analyzes the effect of many users upon an application. Conscientious load testing during the development process protects the application from failing under the weight of simultaneous users. Applied early in the development process, load testing gives developers an opportunity to correct problems—even if it means changing an architecture or major component—before

the problems become virtually too expensive to change.

Once deployed, it is crucial that an application works without interruptions and delays. Load testing charts response time, enabling a developer to recognize those facets of a program that hinder performance. As well, load testing ensures scalability, paving the way for a company's growth.

Pilot testing (otherwise known as beta testing) is the dress rehearsal before the show, the last step before full-scale deployment. It is the determination and culmination of a long, arduous development process. In a pilot test, an application is used by a subset of real users. These users exercise the common portions of the application and determine its effectiveness in helping them conduct business more competitively. Applications that have not been adequately load tested often fail during the pilot stage—and consequently crumble during full-scale deployment.



Load testing

Load testing is a “no-brainer”

Load testing is an essential step in your application development process. It affirms the success of your application by eliminating guesswork about quality and performance. By emulating multiple users simultaneously accessing an application, it quantifies the number of users your application can support and shields your company against slow response time and application failure.

Even the most organized and well thought-out development is certain to have inherent weaknesses. In complex applications, these weak spots are often hidden. Load testing ferrets out these defects and shows you how to correct them.

Without load testing, you can merely surmise about the performance of your application under the stress of many users. Why make an assumption when you can be certain? With load testing, you ascertain the inevitable problems embedded in your application during the development process, when the comparative cost of change is minimal.

Knowledge is power

The knowledge that arises from a load test helps you eliminate guesswork. By emulating many users, you determine, first and foremost, if your application will support its intended audience.

Time is money

Second, you pinpoint the time each user must wait for his or her screen response. Competitiveness depends on your company's ability to do things faster and more efficiently.

Change is inevitable

Third, you are primed for growth. The growing pains that naturally accompany success need not extend to your application. Load testing enables you to check and maintain your application as it grows, so you know if and when to adjust accordingly.

**If you think you don't need load testing, keep in mind the following risks.
Few businesses can afford them.**

- The cost to repair, replace and rework your application after deployment usually exceeds the cost of testing the application.
- An application deployed prematurely causes work stoppages. When a mission-critical application halts, a company loses precious time, revenue and customers.



Benefits of EMPOWER

EMPOWER your application with multi-user quality, performance and scalability

EMPOWER ensures that your application will run—and run smoothly.

EMPOWER secures application performance for today and positions your business for tomorrow.

EMPOWER finds application problems before they become dangerous. It guarantees the productivity of your employees. It prepares your business for change and growth.

Multi-user quality

Using one UNIX machine to simulate hundreds of users, the EMPOWER line of testing products creates scripts to represent actual users and their daily, often disparate, operations. EMPOWER captures user activities—including the actual keystrokes, mouse movements and SQL requests that your employees generate while using an application—and creates emulation scripts. Then, EMPOWER arranges a mix of scripts to represent the many users your application must sustain.

EMPOWER reveals if, and to what degree, your application works. Moreover, it enables you to fix intrinsic problems before deployment.

You can correct difficulties that occur when

- ▶ application code competes with other code for resources
- ▶ access to data is simultaneous
- ▶ your database and OS run out of resources
- ▶ communication subsystems falter

Performance

Hardware, software, database and middleware vendors all laud the speed of their products.

EMPOWER verifies those claims for your primary transactions, at your peak hours of business. When you test your application with EMPOWER, you know the exact time your users must wait for critical responses, and where application errors occur.

Bottlenecks become easy to find and simple to correct. When response time does not meet requirements, developers can react quickly and calmly to solve the problem.

EMPOWER permits them to

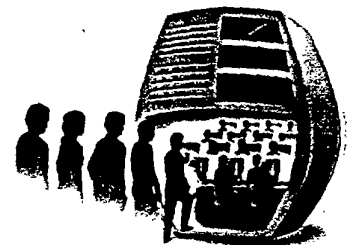
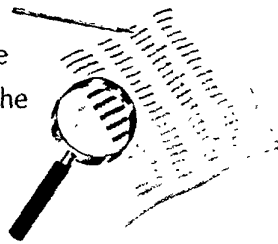
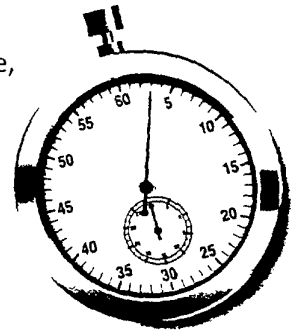
- ▶ enhance application code
- ▶ repartition code and data between client and server
- ▶ optimize database parameters
- ▶ size the CPUs and memory
- ▶ adjust disk layout

Scalability

Likely, your business will grow and change.

Already, you have

invested money and effort in an application to fulfill your present needs. Why not confirm its ability to sustain progress? EMPOWER determines the outcome, with respect to quality and response time, when your



00920746675500



application is forced to support a greater load. You can alter the user level, transaction mixes and rates, and complexity of the application.

EMPOWER is the only true way to verify both the scalability of components and the scalability of components as they work

together. EMPOWER ensures that you can simply and effectively determine capacity when changing

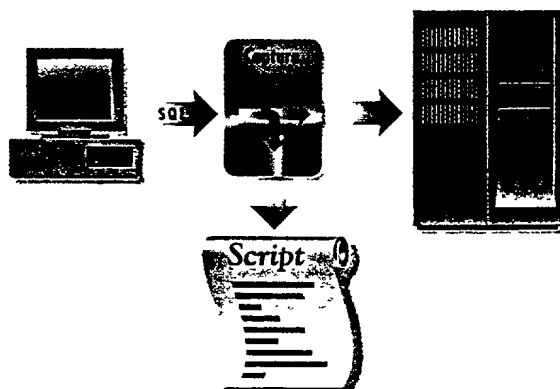
- ▶ workload
- ▶ application code
- ▶ database size
- ▶ hardware configurations
- ▶ software revisions

Script-and-go technology

EMPOWER employs script-and-go technology to build the scripts you need to load test. Script-and-go technology slashes the time needed for script development, making load testing simple and fast. As well, it guarantees the accuracy of your test results by replicating actual user activity.

EMPOWER sits between the user device—whether it be a PC with a client, an X-terminal or simply a VT100—and listens to the traffic between that device and your application. Based on the flow, EMPOWER builds a script that can replace the user device and drive your application.

EMPOWER scripts can be replayed with or without user devices. Running scripts with user devices aides debugging. Running scripts without user devices relieves you from the burden of acquiring a lot of hardware for a multi-user test.



EMPOWER uses the same language for scripts that execute with or without user devices. In fact, there is only one script for which you select the execution mode.

With EMPOWER, you can readily extend captured scripts with loops, randomize script content with if-then-else and switch statements, vary the data in updates and queries, and govern script throughput with thinking, typing and pacing delays. Unique script-and-go technology is streamlined for load testing. It minimizes script development time and maximizes your opportunity for enhancing quality and performance.

The EMPOWER family of products

Improve the quality and performance of your client/server applications with superior load testing software

Performix provides load testing software for a complete myriad of architecture—from simple character-based applications to complex, multi-tier client/server superstructures. The EMPOWER family

of products has tested thousands of applications on a vast array of hardware. It supports all major databases, operating systems, networks and application development tools.

	EMPOWER	EMPOWER/X	EMPOWER/CS
Interface of application to be tested	Character	X-Window	Windows, [™] Windows NT, [™] OS/2, [™] and more
Traffic captured and replayed	Keystrokes	X protocol	Object-oriented events and SQL

Performix facts



Performix, Inc., was founded in 1987 with the vision that load testing would become an integral part of every application development process. In the past eight years, the number of companies embracing load testing has skyrocketed and now includes virtually all businesses with mission-critical applications. Today, Performix is a privately held and funded company with sales and support offices across the United States and in Europe.

All of the premier hardware vendors rely on Performix for their load testing needs, including HP, IBM, Sun, SGI, Sequent, Pyramid, NCR and DEC. Most of these vendors discarded in-house load testing

software in favor of EMPOWER, and saved considerable time and money doing so. The major database vendors, such as Oracle, Sybase and Informix, also depend on the technical prowess of EMPOWER.

Also relying on Performix are leading development companies across all major industries. Some of these prominent customers are AT&T, British Telecom, Sprint, Telia Data, Dow Jones, Fidelity Investments, Standard & Poors, Signet Bank, Mobil, Federal Express, DHL, Hyatt and Computer Associates. As well, major service organizations, such as Andersen Consulting, EDS and CSC, count on Performix to guarantee the success of their projects.

What is load testing?

Load testing is
the process of
analyzing the effect
of many users on
an application.

As part of the application development process, load testing with EMPOWER reveals performance problems before an application is deployed. By emulating users and analyzing the effect of these users on an application, EMPOWER tests the resiliency and flexibility of an application.

- EMPOWER is widely recognized as the best-of-breed among load testing products.
- EMPOWER is available for all types of architecture — from simple character applications to multi-tier client/server superstructures.
- EMPOWER supports all major databases, operating systems, networks and application development tools.
- The EMPOWER family of products has been used to test thousands of applications on a vast array of hardware.

PERFORMIX

Improving the quality and performance of multi-user applications with superior load testing software

PERFORMIX, Inc.
Corporate Headquarters
8200 Greensboro Drive, Suite 1475
McLean, VA 22102-3803 USA
tel 703.448.6606
fax 703.893.1939

Midwest
Naperville, IL
tel 708.305.4261
fax 708.305.4232

New England
Newburyport, MA
tel 508.463.7704
fax 508.463.7705

West
Santa Clara, CA
tel 408.982.8989
fax 408.982.8988

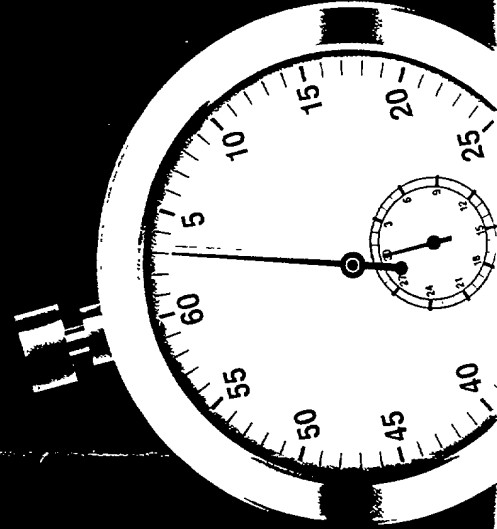
South Central
Dallas, TX
tel 214.774.4460
fax 214.774.4461

PERFORMIX Europe
European Headquarters
The Atrium Court
Apex Plaza
Reading, Berkshire RG1 1AX UK
tel +44.1734.795016
fax +44.1734.795017

Germany
Bad Homburg
tel +49.6172.9258122
fax +49.6172.925889

Copyright ©1995 PERFORMIX, Inc. All rights reserved.
EMPOWER is a registered trademark of PERFORMIX, Inc.
PERFORMIX and EMPOWER/CS are trademarks of
PERFORMIX, Inc. All other trademarks are the property of
their respective holders.

EMPOWER[®] Load Testing Software



As companies depend more heavily on information technology to maintain a competitive edge, application development becomes increasingly rushed and complex. Companies look to their IT departments to build applications that are faster, more reliable and more flexible than ever before — in less time.

In response, developers are searching for new ways to assure the reliability, performance and scalability of their mission-critical applications. In short, to lessen the risks. Load testing is the answer. A load test provides insight about an application's multi-user behavior, enabling developers to deploy quickly and with confidence.

Performix is dedicated to improving the quality and performance of critical applications with load testing software, enabling companies to be proactive, competitive and prepared for the future. EMPOWER software allows tests to be developed and executed with greater speed, at lower cost, and with greater functionality than ever before.

What is EMPOWER?

EMPOWER is load

Benefits of EMPOWER

EMPOWER your application with multi-user quality, performance and scalability.

Multi-user quality

By simultaneously emulating multiple users, load testing determines, first and foremost, if your application will support its intended audience.

Using one machine to simulate hundreds of users, the EMPOWER line of testing products creates scripts that represent actual users and their daily, often disparate, operations. With a capture agent, EMPOWER records user activities — including keystrokes, mouse movements and SQL requests — to create emulation scripts. EMPOWER then arranges a mix of scripts that represents your users.

EMPOWER reveals if, and to what degree, your application works. Before deployment, you can correct common difficulties that emerge during the application development process.

Performance

Load testing charts the time you must wait for screen responses. It finds hidden bugs and bottlenecks and gives you the chance to correct them.

All hardware, software, database and middleware vendors praise the speed of their products. EMPOWER shows you how fast they work within your prescribed environment, for your primary transactions, during your peak hours of business. When you test your applications with EMPOWER, you know the exact time your users must wait for critical responses, and where application errors occur.

Bottlenecks become easy to find and efficient to correct. When response time does not meet requirements, developers can react quickly and calmly to solve problems.

Scalability

Load testing enables you to check and maintain your application as your workload increases, so you can adjust your system accordingly.

Likely, your business will grow and change. Already, you have invested money and effort in an application to fulfill your present needs. Make sure you confirm its ability to sustain progress. When your application is forced to support a

increased workload, EMPOWER determines the outcome, regarding quality and response time. You can alter the user level, transaction mixes and rates, and complexity of the application.

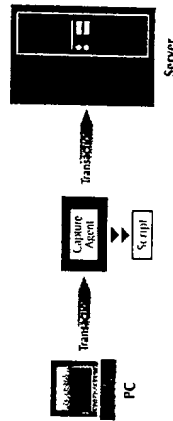
EMPOWER is the only way to verify both the scalability of components and the scalability of components as they work together.

Script-and-Go Technology

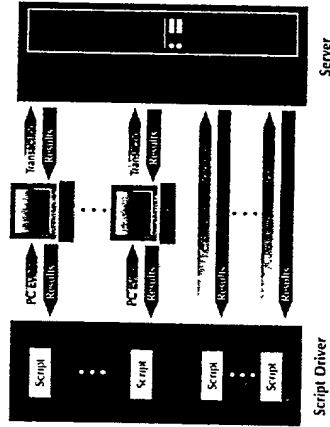
Performix script-and-go technology captures user activity, builds scripts and replays these scripts against both client and server.

The most important feature of script-and-go technology is its capture agent. By definition, this component guarantees the accuracy of any load test. For EMPOWER/CS, the capture agent seizes SQL transactions as they leave the PC on their way to the server. During replay, these same "freeze-framed" transactions test the server. Since there is no proven, foolproof method to recreate communication between a PC and server without a capture agent, EMPOWER/CS is the only solution that provides valid results.

▼ Capture with EMPOWER/CS



▼ Load test with EMPOWER/CS



Deploy applications with ...

- Efficient SQL statements
- Highly-tuned databases
- Properly-sized hardware
- Optimal data and code partitioning
- Adequate database and OS resources
- Data integrity during concurrent access

The beauty of EMPOWER scripts is that a developer can put loops around the transactions to extend the captured activity and change data in the transactions to make multiple iterations more realistic. With simple programming changes, the replayed stream will represent exactly the PC requests that would be generated if an end-user continuously operated your application.

About Performix

Founded in 1987, Performix develops, markets and supports load testing software for companies developing medium to large, mission-critical applications. All of the premier hardware and database vendors rely on Performix to measure and improve their products, and to demonstrate their applications' capacity under heavy user load. Performix was the first company to develop load testing software for X-Windows and the first company to release a client/server load testing product. Today, Performix is a privately held and funded company with headquarters in McLean, Va.